



# High C/C++<sup>™</sup> Language Reference

---

92G6923

**First Edition** (August 1995)

This edition of the *IBM High C/C++™ Language Reference* applies to IBM High C/C++ version 1.0 and to subsequent versions until otherwise indicated in new versions or technical newsletters.

**The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS MANUAL “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.**

IBM does not warrant that the products in this publication, whether individually or as one or more groups, will meet your requirements or that the publication or the accompanying product descriptions are error-free.

This publication could contain technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or program(s) described in this publication at any time.

It is possible that this publication may contain references to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country. Any reference to an IBM licensed program in this publication is not intended to state or imply that you can use only IBM's licensed program. You can use any functionally equivalent program instead.

No part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the written permission of IBM.

Requests for copies of this publication and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

Address comments about this publication to:

IBM Corporation  
Department H83A  
P.O. Box 12195  
Research Triangle Park, NC 27709

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Copyright © 1995, MetaWare® Incorporated, Santa Cruz, CA

©Copyright International Business Machines Corporation 1995. All rights reserved.

Printed in the United States of America.

4 3 2 1

Notice to U.S. Government Users—Documentation Related to Restricted Rights—Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corporation.

## **Patents and Trademarks**

IBM may have patents or pending patent applications covering the subject matter in this publication. The furnishing of this publication does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 208 Harbor Drive, Stamford, CT 06904, United States of America.

IBM and the IBM logo are registered trademarks of the International Business Machines Corporation.

All MetaWare product names are trademarks or registered trademarks of MetaWare Incorporated.

Other terms which are trademarks are the property of their respective owners.



# Contents

---

	<b>About This Book.....</b>	<b>xi</b>
<b>1</b>		
	<b>Introduction to High C/C++.....</b>	<b>1</b>
	1.1 Differences Between ANSI C and C++ .....	1
<b>2</b>		
	<b>C++ Example .....</b>	<b>3</b>
<b>3</b>		
	<b>Lexical Elements of C and C++.....</b>	<b>15</b>
	3.1 Character Set .....	15
	3.2 Punctuators .....	16
	3.3 Whitespace .....	16
	3.4 Comments.....	17
	3.5 Identifiers.....	18
	3.6 Keywords.....	18
	3.7 Operators and Their Precedence.....	19
	3.7.1 Unary Operators .....	23
	3.7.2 Binary Operators.....	24
	3.7.3 Ternary Operators.....	26
	3.8 Constants .....	26
	3.8.1 Decimal Constants.....	26
	3.8.2 Floating-Point Constants .....	27
	3.8.3 Octal Constants.....	28
	3.8.4 Hexadecimal Constants .....	28
	3.8.5 Character Constants.....	29
	3.8.5.1 Escape Sequences .....	30
	3.8.6 Strings.....	31
	3.9 Trigraphs .....	32
<b>4</b>		
	<b>Preprocessing C and C++</b>	
	<b>Program Text.....</b>	<b>33</b>
	4.1 Line Splicing .....	33
	4.2 Preprocessor Directives .....	35

4.2.1	#include and #c_include .....	35
4.2.2	#define and #undef .....	36
4.2.3	Conditional Directives .....	38
4.2.3.1	#if .....	39
4.2.3.2	#ifdef and #ifndef .....	39
4.2.3.3	#elif and #else .....	39
4.2.3.4	Unary Operator defined .....	40
4.2.4	#line .....	40
4.2.5	#error and #print .....	40
4.2.6	#pragma .....	41
4.3	Predefined Macro Names .....	42
4.4	Concatenation .....	42
4.4.1	Uses of Operators # and ## .....	43

## 5

<b>Declarations and Definitions .....</b>	<b>47</b>
5.1 Declarations .....	48
5.2 Blocks and Name Scoping .....	49
5.3 Storage Categories .....	50
5.3.1 auto Storage .....	51
5.3.2 extern Storage .....	51
5.3.3 register Storage .....	51
5.3.4 static Storage .....	51
5.3.5 Default Storage Categories .....	52
5.4 Data Types .....	52
5.4.1 Type Specifiers .....	54
5.4.2 Scope Access .....	55
5.4.3 Fundamental Types .....	56
5.4.4 Type enum .....	58
5.4.5 Derived Types .....	59
5.4.5.1 Pointers .....	60
5.4.5.2 References .....	60
5.4.5.3 Arrays .....	61
5.4.5.4 Structures .....	62
5.4.5.5 Unions .....	64
5.4.5.6 Bit Fields .....	66
5.4.6 Type Qualifiers .....	67
5.4.6.1 Type Qualifier const .....	67
5.4.6.2 Type Qualifier volatile .....	68

5.4.6.3	Type Qualifiers <code>_Alias</code> and <code>_Noalias</code> .....	69
5.4.7	Defining Types with <code>typedef</code> .....	69
5.5	Declarations and Precedence.....	70
5.6	Linkage Specifications .....	71
5.7	Initialization.....	72

## 6

<b>Expressions and Conversions</b> .....	<b>77</b>
6.1 Elements of an Expression .....	77
6.2 Conditional Expressions.....	78
6.3 Constant Expressions .....	78
6.4 Operators .....	79
6.4.1 Assignment .....	79
6.4.2 Comma.....	81
6.4.3 Addition and Subtraction.....	81
6.4.4 Multiplication, Division, and Modulus.....	82
6.4.5 Relational Operators .....	83
6.4.6 Shift Operators.....	84
6.4.7 Logical AND and OR .....	85
6.4.8 Bitwise AND, OR, and Exclusive OR.....	85
6.4.9 Pointer-to-Member Operators.....	86
6.4.10 Unary Operators .....	87
6.4.10.1 Address Operator .....	87
6.4.10.2 Dereference Operator .....	87
6.4.10.3 Unary Plus .....	87
6.4.10.4 Unary Minus .....	88
6.4.10.5 Bitwise NOT.....	88
6.4.10.6 Logical NOT.....	88
6.4.10.7 Pre-Increment and Pre-Decrement .....	88
6.4.10.8 Post-Increment and Post-Decrement .....	89
6.4.10.9 The <code>sizeof</code> Operator .....	90
6.4.10.10 The <code>delete</code> Operator .....	90
6.4.10.11 The <code>new</code> Operator .....	91
6.5 Conversions .....	93
6.5.1 Floating-Point Conversions .....	93
6.5.2 Integer Conversions.....	94
6.5.3 Arithmetic Conversions.....	94
6.5.4 Pointer Conversions.....	95
6.5.5 Type Casting.....	96

## 7

<b>Statements</b>	<b>99</b>
7.1 Labeled Statements	100
7.2 Expressions as Statements	100
7.3 Compound Statements or Blocks	101
7.3.1 Resolving Ambiguity	101
7.4 Selection Statements	102
7.4.1 switch Statement	102
7.4.2 if Statement	104
7.5 Iteration Statements	105
7.5.1 do Statement	105
7.5.2 while Statement	106
7.5.3 for Statement	106
7.6 Jump Statements	108
7.6.1 break Statement	109
7.6.2 continue Statement	109
7.6.3 return Statement	109
7.6.4 goto Statement	110

## 8

<b>Functions</b>	<b>111</b>
8.1 Recursive Functions	113
8.2 Nested Functions	113
8.3 The Function main	114
8.4 inline Functions	115
8.5 friend Functions	116
8.6 Function Parameters	116
8.6.1 Passing Parameters by Value and by Reference	117
8.6.2 Reference Parameters	118
8.6.3 Parameter Evaluation Order	119
8.6.4 Variable-Length Parameter Lists	119
8.6.5 Default Parameters	121

## 9

<b>C++ Classes</b>	<b>123</b>
9.1 Class Declarations and Definitions	123
9.2 Local Classes	127
9.3 Nested Classes	127
9.4 Class Members	128



9.4.1	Simple Members.....	128
9.4.2	typedef Members .....	131
9.4.3	static Members.....	131
9.5	Member Functions.....	132
9.5.1	static Member Functions .....	133
9.5.2	Non-static Member Functions .....	134
9.5.3	Qualified Member Functions.....	136
9.6	Derived and Base Classes.....	137
9.7	Virtual Functions .....	141
9.8	Abstract Classes .....	146
9.9	Virtual Base Classes .....	146
9.10	Pointers to Members.....	148
<b>10</b>		
	<b>Access Control in C++ .....</b>	<b>151</b>
10.1	Access.....	151
10.2	Access Override.....	157
10.3	Multiple Access Through Virtual Base Classes .....	158
10.4	Friends.....	159
<b>11</b>		
	<b>Special C++ Member Functions.....</b>	<b>163</b>
11.1	Constructors.....	165
11.1.1	Constructor Initializers .....	170
11.1.2	Virtual Base Classes and Constructors.....	172
11.1.3	Copy Constructors .....	174
11.2	Destructors.....	176
11.3	User-Defined Conversions .....	179
11.4	Operators new and delete .....	184
11.5	Assignment Operator = .....	186
<b>12</b>		
	<b>Name Overloading and Operator Functions in C++ .....</b>	<b>191</b>
12.1	Overload Resolution.....	191
12.1.1	Argument-Match Overload Resolution .....	192
12.1.2	Argument Type Matching .....	194
12.1.3	Function-Match Overload Resolution .....	196
12.2	Overloading and Scope.....	197
12.3	Operator Functions .....	198
12.3.1	Binary Operators.....	201

12.3.2	Prefix Unary Operators .....	201
12.3.3	Postfix Unary Operators ++ and -- .....	201
12.3.4	Postfix Function-Call Operator ( ) .....	202
12.3.5	Postfix Subscript Operator [ ] .....	203
12.3.6	Postfix Dereference Operator -> .....	203
<b>13</b>		
	<b>C++ Templates .....</b>	<b>205</b>
13.1	Template Type Equivalence.....	207
13.2	Function Templates.....	208
13.3	Template Declarations and Definitions.....	210
13.4	Template Member Functions .....	211
13.5	Template Friends .....	212
13.6	Static Members and Variables in a template .....	212
<b>14</b>		
	<b>C++ Namespaces .....</b>	<b>215</b>
14.1	When to Use Namespaces.....	215
14.2	How to Specify a Namespace .....	216
14.3	Using the Names Declared in a Namespace .....	217
14.3.1	The using Declaration.....	218
14.3.2	The using Directive.....	220
14.3.3	Namespaces, friends and externs .....	222
14.3.4	Unnamed Namespaces .....	222
14.4	Current Status of Namespaces .....	223
14.5	Suggested Reading.....	224
<b>15</b>		
	<b>High C++ Exception Handling .....</b>	<b>225</b>
15.1	Detecting Errors .....	225
15.2	How Exception Handling Works .....	225
15.2.1	The throw, try, and catch Statements .....	225
15.2.2	What Happens When an Exception Is Thrown .....	227
15.3	Unwinding: Automatic Clean-Up with Destructors .....	228
15.3.1	Additional Clean-Up During Unwinding.....	229
15.3.2	Stacked and Nested Exceptions .....	231
15.4	Exceptions During Unwinding: the terminate() Function.....	232
15.5	Partially Constructed and Destroyed Heap Objects.....	233
15.6	Function Exception Specifications .....	234
15.7	Miscellaneous Rules for Handling Exceptions .....	236

15.8	What a Handler Can Catch .....	236
15.9	Suggested Reading .....	237
<b>16</b>		
	<b>C++ Run-Time Type Information (RTTI) .....</b>	<b>239</b>
16.1	RTTI and typeid .....	239
16.2	The Type Object .....	240
16.2.1	Using the Type Object .....	242
16.2.2	Explicit Initialization in Conditions .....	243
16.3	Functions on Type Objects .....	244
<b>17</b>		
	<b>New Cast Notation for C++ .....</b>	<b>247</b>
17.1	static_cast<Type>(expr) .....	247
17.2	const_cast<Type>(expr) and Casting Away const ...	248
17.3	dynamic_cast<Type>(expr) .....	249
17.4	reinterpret_cast<Type>(expr) .....	250
<b>A</b>		
	<b>Migrating from C to C++ .....</b>	<b>251</b>
A.1	C/C++ Compatibility Issues .....	251
A.2	Using Incremental C++ .....	252
A.2.1	Using Special Keywords and Functions .....	253
A.2.2	Accessing C++ Features Incrementally .....	254
A.3	C++ Features You Cannot Use in Incremental Mode .....	257
A.4	Moving from Incremental C++ to Full C++ .....	257
<b>B</b>		
	<b>High C/C++ Extensions .....</b>	<b>259</b>
B.1	Underscores in Numbers .....	259
B.2	Significant Characters in Identifiers .....	259
B.3	Case Ranges in switch Statements .....	259
B.4	Initializing Automatic Aggregates .....	260
B.5	Reserved Word pragma .....	260
B.6	Named Parameter Association .....	261
B.6.1	Named Parameter Syntax .....	261
B.6.2	Constraints on Using Named Parameters .....	262
B.7	Nested Functions and Full-Function Variables .....	263
B.7.1	Up-Level Referencing .....	263
B.7.2	Full-Function Values .....	264

B.7.3	Non-Local Labels .....	266
B.8	Intrinsics .....	268
B.9	Packed and Unpacked Structures .....	268
B.10	Near and Far Objects and Pointers .....	268
B.10.1	Distance Attributes .....	269
B.10.2	Type and Assignment Compatibility .....	270
B.11	Differences Between High C Syntax and Others .....	271
B.12	Predefined Macro <code>__HIGHC__</code> .....	272
B.13	Calling-Convention Attribute Specified with Function Declaration . 273	
B.13.1	Kanji Support .....	274
B.14	<code>short enum</code> and <code>long enum</code> .....	275
B.15	Arbitrary Base for Numbers .....	275
B.15.1	ANSI Preprocessor Numbers .....	276
B.15.2	Preprocessor Directives <code>#assert</code> and <code>#unassert</code> ....	276

## C

	<b>Iterators in High C/C++ .....</b>	<b>279</b>
C.1	What Is an Iterator? .....	279
C.2	How Do Iterators Work? .....	280
C.3	Some Uses for Iterators .....	281
C.4	Recursion and Code Clarity .....	283
C.5	Replacing Macros with Iterators .....	285
C.6	Iterator Syntax and Constraints .....	288
C.6.1	Arguments to Iterators .....	288
C.6.2	Predefined Function <code>yield()</code> .....	289
C.7	Semantics of Iterators .....	289
C.8	Advantages and Disadvantages of Iterators .....	292
	<b>Index .....</b>	<b>295</b>

# Tables

---

Table 3.1	Punctuators.....	16
Table 3.2	Keywords in High C++.....	19
Table 3.3	Precedence and Associativity of C++ Operators.....	21
Table 3.4	Unary Operators.....	23
Table 3.5	Binary Operators.....	24
Table 3.6	Escape Sequences .....	30
Table 3.7	Trigraphs in C++.....	32
Table 4.1	Preprocessor Directives in High C/C++ .....	35
Table 4.2	Predefined Macro Names.....	42
Table 5.1	C and C++ Predefined Data Types .....	56
Table 5.2	Minimum Ranges for Integers .....	57
Table 6.1	C++ Relational Operators .....	83
Table 6.2	Arithmetic Conversions .....	94
Table 7.1	C++ Statements.....	99
Table 10.1	Access of Members of Base and Derived Classes.....	154
Table 11.1	C++ Special Members .....	164
Table 16.1	Objects Returned by typeid Expressions.....	240
Table A.1	Incremental C++ Keywords.....	253

# Figures

---

Figure 2.1	Inheritance of Data Members and Member Functions .....	9
Figure 2.2	Static Data Members .....	12
Figure 16.1	A Polymorphic typeid .....	241

# About This Book

---

This manual describes the C++ language and MetaWare High C/C++ extensions. We discuss the syntax and semantics of C++, and highlight differences between C++ and C. We assume you are familiar with the C programming language and C++ programming concepts.

---

## Related C/C++ Publications

For information about the C language, consult the following:

ANSI Committee X3J11. **American National Standard for Information Systems — Programming Language C**. Doc. No. X3.159. CBEMA, 1989.

Plauger, P. J., and J. Brodie. **Standard C**. Microsoft Press, 1989.

For information about C++, see:

Stroustrup, B. **The C++ Programming Language**, second edition. Addison-Wesley Publishing Company, 1991.

Stroustrup, B. and Ellis, M. **The Annotated C++ Reference Manual**. Addison-Wesley Publishing Company, 1990.

For an overview of object-oriented programming, you may find the following book helpful:

Booch, G. **Object-Oriented Design with Applications**. Benjamin/Cummings Publishing Company, Inc., 1991

---

## Related MetaWare Publications

To learn how to use the MetaWare High C/C++ compiler, and for system-specific information, see the **High C/C++ Programmer's Guide**.

If MetaWare run-time libraries are supplied with your compiler, see the **High C Library Reference Manual** for informations about C functions, and the

**High C++ I/O Streams Library Reference** for information about streams classes and member functions.

---

## Notational and Typographic Conventions

Several notational and typographic conventions are used in this manual to visually differentiate text.

<i>Courier</i>	Program text, input, output, and file names.
<b><i>Courier</i></b>	Commands, keywords, command-line options, literal options, and preprocessor directives.
<i>Italic Courier</i>	Formal parameters and template values to be replaced by user-specified names or values.
{ }	Select one and only one of the options separated by vertical bars and enclosed in curly braces.
[ ]	Select none or one of the options separated by vertical bars and enclosed in square brackets.
	Vertical bars separate choices within brackets or braces.
...	Repeatedly (one or more times) select an instance of the preceding item.
	MetaWare extension to C and/or C++.
	C++ feature not available in ANSI Standard C.
	Semantic difference between ANSI Standard C and C++.

**Extension**

**C++ Only**

**C ≠ C++**



# Introduction to High C/C++

---

The High C/C++ compiler conforms to the American National Standards Institute (ANSI) X3J11/90-013 standard for the C programming language. In addition, MetaWare implements all of the C++ programming language as defined in **The Annotated C++ Reference Manual** by Margaret Ellis and Bjarne Stroustrup. MetaWare is participating in the ANSI committee's formal definition of the C++ language and has incorporated language changes to track the changing ANSI X3J16 standard.

High C/C++ includes many extensions not found in other C and C++ products. For information about these extensions, see Appendix B: *High C/C++ Extensions* and Appendix C: *Iterators in High C/C++*.

---

## 1.1 Differences Between ANSI C and C++

C++ is not quite a superset of ANSI Standard C. Some of the important differences are listed below. If you have been programming in C, you should keep these differences in mind when you write C++ programs or add C++ features to existing C programs. See Appendix A: *Migrating from C to C++* for more information.

- The following words can be used as identifiers in ANSI Standard C, but not in C++, where they are reserved (see §3.6: *Keywords* on page 18):

<b>catch</b>	<b>mutable</b>	<b>protected</b>	<b>try</b>
<b>class</b>	<b>namespace</b>	<b>public</b>	<b>typeid</b>
<b>delete</b>	<b>new</b>	<b>template</b>	<b>using</b>
<b>friend</b>	<b>operator</b>	<b>this</b>	<b>virtual</b>
<b>inline</b>	<b>private</b>	<b>throw</b>	

- In C++, a **struct** defines a type. The following code is valid in C++ but not in ANSI Standard C (see §5.4.5.4: *Structures* on page 62):

```
struct X { int n; };  
X a1;
```

- In C++, a **struct** defines a scope; in ANSI Standard C a **struct**, as well as an enumeration or an enumerator declared in a **struct**, is exported to the enclosing scope (see §5.4.2: *Scope Access* on page 55).
- In C++, a **void\*** can be assigned only to another **void\***. In ANSI Standard C a **void\*** can be assigned to a pointer of any type (see §6.5.4: *Pointer Conversions* on page 95).
- In ANSI Standard C, a global **const** has external linkage by default; in C++ it does not. (see §5.4.6.1: *Type Qualifier const* on page 67).
- Character constants are **char** types in C++, **int** types in ANSI Standard C (see §3.8.5: *Character Constants* on page 29).
- The type of an enumerator is the type of its enumeration in C++; in ANSI Standard C it is an **int** (see §5.4.4: *Type enum* on page 58).

# 2

## C++ Example

---

- Important features of C++* This chapter presents a C++ program called `example.cpp` that illustrates some important features of the language. These features are discussed briefly and cross-referenced to more detailed information in later chapters. Experienced C programmers might want to examine `example.cpp` in detail, and then skip to sections of the manual that discuss unfamiliar features of C++.
- Source code included* The full source for `example.cpp` is available in your High C/C++ distribution. Here we present and discuss `example.cpp` in a series of code fragments. This simple program does the following:
- Establishes an employee database and reads some predefined records from a data file `example.dat`
  - Enables you to select and view record headers or entire records
  - Enables you to add or delete records

### Example 2.1 Part 1

---

```
// File example.cpp - C++ example program
// Copyright MetaWare Incorporated 1992
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
// Field-Length constants:
const Choice_len = 10;
const Agency_len = 30;
const City_len   = 15;
const Date_len   = 10;
const Dept_len   = 30;
const F_N_len    = 10;
const L_N_len    = 15;
const Phone_len  = 15;
const SSN_len    = 15;
const State_len  = 20;
const Status_len = 10;
const Street_len = 30;
```

```
const Title_len  = 30;
const Type_len   = 10;
// Function prototypes:
int get_num (int&);
int get_num (long&);
int get_num (float&);
void get_line (char*,int);
void menu ();
```

---

C++ comment style,  
*const*, and  
prototyping

Example 2.1: *Part 1* demonstrates the C++ comment style. Double forward slashes comment text to the end of the line. Like ANSI Standard C, C++ uses the keyword **const**. **const** is the preferred way to define constants, and is meant to replace the **#define** preprocessor directive in most cases. See §5.4.6.1: *Type Qualifier const* on page 67 for more information about **const**. Also note the use of function prototypes, a feature familiar to ANSI Standard C users. Function prototypes are required in C++. Prototyping makes programs easier to read and maintain.

I/O streams

The **iostream.h** and **fstream.h** header files contain stream library information that C++ needs to perform standard I/O and file I/O, respectively. Streams are not files, but you can think of them as similar to file pointers in ANSI Standard C. See the **High C++ I/O Streams Library Reference** for more information about streams.

## Example 2.1 Part 2

---

```
class employee {
protected:
    char *Soc_Sec_Num;
    char *Home_Phone;
    char *Street;
    char *City;
    char *State;
    char *Type;
    char *Last_Name;
    char *First_Name;
    char *Title;
    char *Department;
    char *Work_Phone;
    char *Start_Date;
```

## 2: C++ Example

```
void print_general ();  
virtual void getinfo_spec ()=0;  
void getinfo_general ();  
public:  
    employee ();  
    virtual ~employee ();  
    void print_header ();  
    virtual void print_report ()=0;  
};
```

---

A **class** is a  
**struct** with  
functions

C++ introduces a new concept known as a *class*. In C++, classes, structures, and unions can all contain both functions and data. In effect, a class is simply a **struct** with functions. Classes provide a more direct association between code and data than is possible in ANSI Standard C. Example 2.1: *Part 2* declares **class employee**.

Class objects

Just as C **struct** types do, classes introduce new types in C++. A *class object* is an object whose type is a class type. This is sometimes referred as an *instance* or *instantiation* of a class. In C++, *object-oriented programming* is accomplished by dealing with class objects.

Member functions

Functions declared in a class are called *member functions* of that class. For example, **print\_general()** and **getinfo\_spec()** are member functions of **employee**. Member functions have special *access rights* to class data.

Access rights:  
**public**, **private**,  
**protected**

The **employee** class introduces three new C++ keywords: **public**, **private**, and **protected**. These keywords determine access to class members.

- **public** members (those whose declarations follow the keyword **public**) are available to any function.
- **private** members can be accessed *only* by members of the same class.
- **protected** members can be accessed by member functions of the same class and by certain other classes called *derived* classes.

Encapsulating data

The practice of allowing only certain functions to access a data structure is called *encapsulation*. See Chapter 10: *Access Control in C++* for more information about access control.

**Example 2.1 Part 3**

---

```

employee::employee () {
    // This constructor allocates memory for class members.
    Last_Name      = new char [L_N_len];
    First_Name     = new char [F_N_len];
    Title          = new char [Title_len];
    Department     = new char [Dept_len];
    Work_Phone     = new char [Phone_len];
    Soc_Sec_Num    = new char [SSN_len];
    Home_Phone     = new char [Phone_len];
    Street         = new char [Street_len];
    City           = new char [City_len];
    State          = new char [State_len];
    Type           = new char [Type_len];
    Start_Date     = new char [Date_len];
};

employee::~~employee () {
    // This destructor frees memory allocated by a
    // constructor. This destructor is implicitly
    // called by the destructors for classes
    // temp and permanent.
    delete Last_Name;
    delete First_Name;
    delete Title;
    delete Department;
    delete Work_Phone;
    delete Soc_Sec_Num;
    delete Home_Phone;
    delete Street;
    delete City;
    delete State;
    delete Start_Date;
};

```

---

The first **public** member function in `employee` is a special function known as a *constructor*. A constructor is called in the following circumstances:

- when the program executes the declaration of an object
- when you create an object with **new**

## 2: C++ Example

*Constructors initialize objects* A constructor guarantees that your object is initialized before you use it. In this example, the constructor `employee()` allocates arrays of characters to hold information about the employee. Each constructor has the same name as its class. A constructor that can be called with no arguments is a *default constructor*. The constructor defined in the `employee` class is a default constructor.

*Destructors destroy objects* The `employee` class contains a constructor and a *destructor*. A destructor does everything necessary to destroy a particular class object, causing that object not to exist. A destructor's name is the name of the class, prefixed with the `~` character. For example, `~employee` frees up the memory allocated by constructor `employee()`. A destructor for an object is called in the following circumstances:

- when the program leaves the object's scope
- when you delete an object with `delete`

See Chapter 11: *Special C++ Member Functions* for more information about constructors and destructors.

*Managing storage with new and delete* C++ introduces two new functions to deal with storage allocation: `new` and `delete`. `new` calls the constructor to allocate memory for that object. `delete` destroys an object created by `new` and frees up the storage. `delete` invokes the destructor (if any) for the object pointed to. See §11.4: *Operators new and delete* on page 184 for more information about storage allocation.

### Example 2.1 Part 4

---

```
class temp : public employee {
private:
    void getinfo_spec ();
    float Hourly_Rate;
    char *Agency;
    int Contract_Duration;

public:
    temp ();
    temp (ifstream&);
    ~temp () { delete Agency; };
    void print_report ();
};
```

```

class permanent : public employee {
private:
    int Medical_Coverage;
    long Annual_Salary;
    void getinfo_spec ();
public:
    permanent ();
    permanent (ifstream&);
    ~permanent () {};
    void print_report ();
};

```

---

*Derived classes  
inherit members  
from base classes*

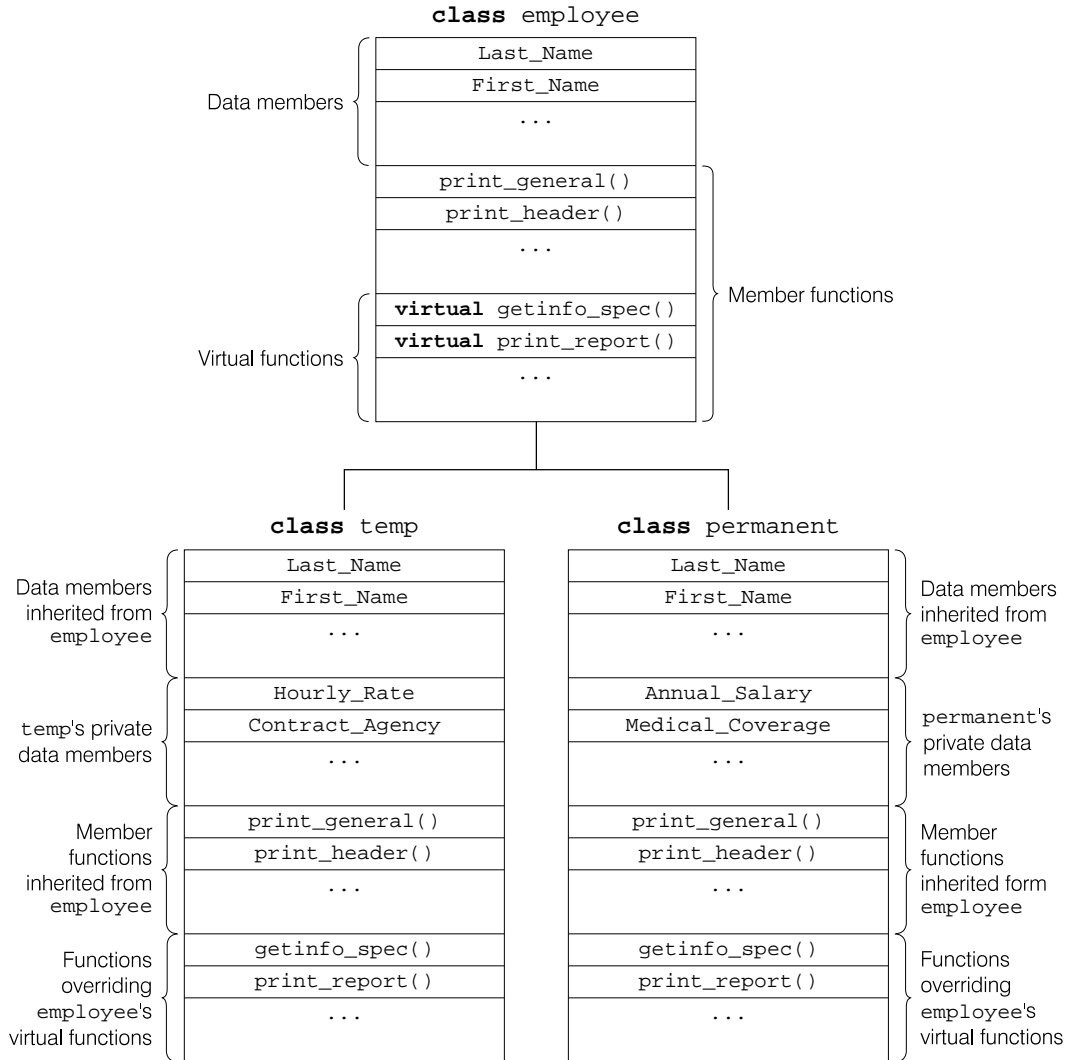
Example 2.1: *Part 4* illustrates the classes `temp` and `permanent`. These classes represent data for temporary and permanent employees, respectively. As in the real world, “temps” and “permanents” are types of employees. These classes inherit the attributes of generic employees (name, address, and so on), but add their own specific attributes. Class `employee` is a *base class* and classes `temp` and `permanent` are *derived classes*. They inherit the **protected** and **public** members of class `employee`.

In addition, the `temp` class adds its own **private** members, such as `Hourly_Rate`, `Agency`, and `Contract_Duration`. Similarly, the `permanent` class adds its own members such as `Medical_Coverage` and `Annual_Salary`, which apply only to permanent employees (see Figure 2.1: *Inheritance of Data Members and Member Functions* on page 9).

See §9.6: *Derived and Base Classes* on page 137 for more information about inheritance.



## 2: C++ Example



**Figure 2.1** Inheritance of Data Members and Member Functions

**Example 2.1 Part 5**

---

```

class Node {
private:
    Node *next;
    employee *empl;
    static int records; // static class member:
                        // number of Nodes in list
public:
    // Default constructor:
    Node () { empl = 0; next = 0; };
    // Second constructor with inlined code:
    Node (employee *new_emp) {
        empl = new_emp;
        next = 0;
    };
    // Overloaded operators:
    void operator+= (employee*);
    int operator-= (int);
    // Friend functions:
    friend void headers (Node&);
    friend void view_record (Node&);
};

int Node::records = 0; // Initializes record count
// Function prototypes:
void add (Node&);
void remove (Node&);
void view_record (Node&);
// Overloaded operator +=:
void Node::operator+= (employee *new_emp) {
    // Appends a node to the linked list
    Node *last = this;
    while (last->next)
        last = last->next;
    last->next = new Node (new_emp);
    records++;
};
// Overloaded operator -=:
int Node::operator-= (int choice) {
    // Removes a node from the linked list
    int i=1,result;

```

## 2: C++ Example

```
    if (choice >= 1 && choice <= records) {
        Node *last = this;
        while (i < choice) {
            last = last->next;
            i++;
        };
        Node *hold = last->next;
        if (hold) {
            last->next = (last->next)->next;
            delete hold;
            records--;
        };
        result = 0;
    }
    else result = 1;
    return result;
};

void add (Node& first) {
    // Adds a node to the linked list
    char choice_str[choice_len],choice;
    cout << "temporary or permanent? ";
    get_line (choice_str,choice_len);
    choice = choice_str[0];
    switch (choice) {
        case 't':
        case 'T':
            first += new temp;
            break;
        case 'p':
        case 'P':
            first += new permanent;
            break;
        default:
            break;
    };
};
```

---

Example 2.1: *Part 5* on page 10 introduces a new class, `Node`. `Node` is a linked list of pointers to `employee` objects.

Operator functions  
and operator  
overloading

`Node` declares two operator functions, `+=` and `--`. `+=` is declared like this:

```
void Node::operator +=(employee *new_emp);
```

This function appends the `employee` object to the linked list. The `+=` operator is invoked in `add()` by this line:

```
first += new temp;
```

Operator functions make code easier to read because they convey their meaning more concisely than normal function names. The `--` operator function removes an employee record from the database; `choice` is the record number to delete:

```
first -= choice;
```

Because `+=` and `--` already exist in C++ and operate on other types (`int`, `float`, and so on), these operators are said to be *overloaded*. Overloaded operators have the same name, but are distinguishable by a unique parameter list. Different versions of `+=` and `--` are called, depending on how they are used. See Chapter 12: *Name Overloading and Operator Functions in C++* for more information about operator overloading.

Static data members  
and scope  
resolution

Note this line in the definition of `Node`:

```
static int records; // static class member
```

The keyword `static` indicates that there is exactly one data member `records` for the class `Node` (Figure 2.2: *Static Data Members*). `static` data members are used by objects of a class to communicate with one another. `Node` uses data member `records` to keep a count of the number of nodes in the linked list.

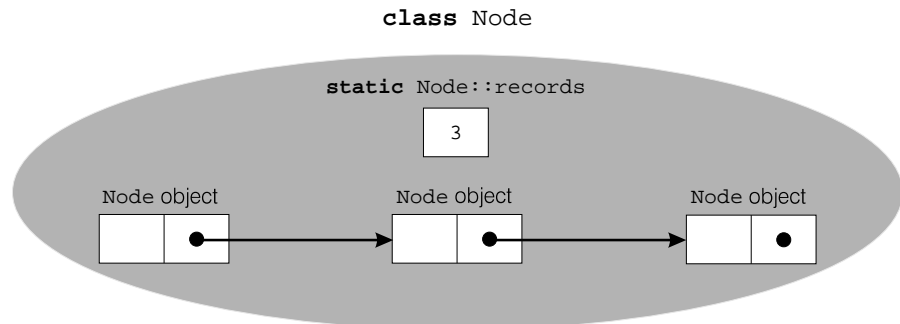


Figure 2.2 Static Data Members

## 2: C++ Example

*Friend functions* Node declares two **friend** functions:

```
friend void headers (Node&);  
friend void view_record (Node&);
```

**friend** functions have the same access rights as member functions, without being declared as members. **friend** functions are used in `example.cpp` to grant non-member functions `headers()` and `view_record()` access to class `Node`. **friend** functions are also useful when one function must operate on two separate classes, and you do not want to duplicate the function. See §10.4: *Friends* on page 159 for more information about **friend** functions.

*Hidden parameter* **this** The += operator uses the hidden **this** parameter:

```
last = this
```

Each member function has a hidden parameter, denoted by the keyword **this**. **this** points to the class object whose members the function manipulates. See §9.5.2: *Non-static Member Functions* on page 134 for more information about **this**.

*Virtual functions and polymorphism* The following line appears in the `employee` class:

```
virtual void print_report()+0;
```

`print_report()` is a *virtual function*. **virtual** function bodies are not inherited from the base class and can be overridden by derived classes. In this case, the `print_report()` function body is empty. **virtual** functions allow this call in function `view_record`:

```
(current->empl)->print_report();
```

This call invokes `print_report()`, but the type of object pointed to by `current->empl` is unknown until run time. At run time, the program determines whether the object is `temp` or `permanent`, and calls the appropriate `print_report()`. This is known as *late binding*, and is one of the most powerful features of C++. The concept of sharing the same name throughout a class hierarchy is known as *polymorphism*. See §9.7: *Virtual Functions* on page 141 for more information about **virtual** functions.

*Abstract base classes and pure virtual functions* The declaration of `getinfo_spec()` in class `employee` may look peculiar:

```
virtual void getinfo_spec ()=0;
```

`getinfo_spec` is set to 0 (zero) to indicate it is a *pure virtual function*. A pure virtual function is not defined, and can belong only to an abstract class. An *abstract class* contains one or more pure virtual functions. Abstract classes, such as `employee`, cannot be instantiated; they can be used only as bases for inheritance. `employee` is used only as a base for the derivation of `temp` and `permanent`, not to create `employee` objects.

# Lexical Elements of C and C++

---

*Tokens* This chapter describes the ways you can use individual characters to form C++ tokens. A *token* is a sequence of characters that constitute a language element. The set of tokens is the vocabulary of the C++ language. Each kind of token is a basic element of the C++ language.

---

## 3.1 Character Set

These are the characters in C++:

- all uppercase and lowercase alphabetic characters `A..Z` and `a..z`
- all numeric characters `0..9`
- space, vertical-tab, horizontal-tab, carriage-return, and line-feed
- the following characters:

`! % ^ & * ( ) - + = { } | ~ [ ]`  
`\ ; ' : " < > ? , . / # _`

These characters can be combined in various ways to form identifiers, operators, strings, comments, numbers, delimiters, and punctuators.

## 3.2 Punctuators

Table 3.1 shows the punctuators valid in High C++.

**Table 3.1 Punctuators**

Punctuator	Description
[ and ]	Indicate array subscripts
( and )	Indicate a function call or group operation in an expression; also used as the beginning and ending marks of parameter lists, and other uses
{ and }	Enclose lists of statements, grouping them to form a compound statement. Enclose lists of initial values. Mark beginning and ending points of member lists for structures, unions, and classes
,	Separates two expressions or items in a parameter list
;	Terminates statements, declarations, and pragmas
:	Used as a separator for case labels, statement labels, bit fields, and constructor initializers
...	Indicates that a function has a variable number of arguments
..	Indicates a range, such as 'a'..'z' in <b>switch</b> statements (High C/C++ extension)

**Extension**

## 3.3 Whitespace

Whitespace refers to a set of characters, such as blanks, tabs, and the newline character, that increase readability of the source program. These characters have no other significance in the program, unless they appear within a literal string or character constant. Comments (see §3.4: *Comments* on page 17) are also whitespace. The following are three whitespace examples.

```

int i  =  3;  // Extra blanks around = do not matter.
int i=3;      // This line is correct also.
inti=        // But this does not declare an integer i.
              // Leaving out the space between int and i
              // changes the "inti" to an identifier.

```



## 3.4 Comments

A *comment* is text used as descriptive commentary; it has no meaning to the program. Comments have no effect when the program is executed.

*Two forms of comments* C++ has two forms of comments. The first form is familiar to C programmers, as shown in the following example:

```
/* Any sequence of characters, except: */
```

This comment starts with the characters `/*` and ends with the characters `*/`. Comments *cannot* be nested, as shown in the following examples.

```
/* This is a valid comment. */
/* This is not /* a valid comment. */ */
```

The second form of a comment in C++ starts with two slashes (`//`) with no space between them, as shown in the following examples:

```
// This whole line is a comment.
main () // This part of the line is a comment.
int i = 3 // Oops, cannot have space between slashes.
```

**C ≠ C++**

The `//` form of comment extends to the end of the line. Using `//`-style comments may lead to a different interpretation in C++ versus ANSI Standard C. For example:

```
int c = q/*divide by 6 */6;
+a;
```

In ANSI Standard C this statement is interpreted as:

```
int c = q / 6; +a;
```

while in C++ the statement is interpreted as:

```
int c = q +a;
```

You can avoid this problem by judicious use of whitespace, as shown in the following example:

```
int c = q // This will not be misinterpreted.
+a;
int c = q /* The compiler will not misinterpret */
+a;           // this, but a person might.
```

*Watch out for line splicing* If a line containing a `//`-style comment ends in a backslash, the next line is appended; see §4.1: *Line Splicing* on page 33 for more information.

## 3.5 Identifiers

An *identifier* is a name for such things as variables, functions, and classes in C++. The form of an *identifier* is:

`{ letter | _ } [ letter | digit | _ ] . . .`

*Letters* A *letter* is any of the following characters:

`a b c d e f g h i j k l m n o p q r s t u v w x y z`  
`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

*Digits* A *digit* is any of the following characters:

`0 1 2 3 4 5 6 7 8 9`

*All characters are significant in an identifier* Keywords (see §3.6: *Keywords*) and library function names are forms of identifiers. An identifier can begin with a letter or underscore, followed by an arbitrarily long sequence of letters, underscores, or digits in any order. Identifiers cannot contain whitespace. All characters of an identifier are significant. Identifiers are case sensitive; `ROSE` is a different identifier from `rose`, and also is different from `roSe`.

*Avoid leading underscores* Avoid using identifiers that begin with one or two underscores in your programs, because these identifiers are reserved for use in the run-time library and in constructing names for member functions.

## 3.6 Keywords

*Reserved identifiers* A *keyword* is an identifier that has a special usage in High C++. Keywords are reserved identifiers, and cannot be redeclared or used in any context that is not valid for the keyword's special use. Table 3.2 shows High C++ keywords.

### Extension

The keywords in Table 3.2 that begin with an underscore are High C/C++ extensions to C and C++: `_Alias`, `_CC`, `__declspec`, `_Dpascal`,

`_Far`,  
`_Near`, `_Noalias`, `_Packed`, and `_Unpacked`.

**Table 3.2** Keywords in High C++

<code>_Alias</code>	<code>default</code>	<code>if</code>	<code>private</code>	<code>throw</code>
<code>_Asm</code>	<code>delete</code>	<code>inline</code>	<code>protected</code>	<code>try</code>
<code>auto</code>	<code>do</code>	<code>int</code>	<code>public</code>	<code>typedef</code>
<code>break</code>	<code>double</code>	<code>long</code>	<code>register</code>	<code>typeid</code>
<code>case</code>	<code>_Dpascal</code>	<code>mutable</code>	<code>return</code>	<code>union</code>
<code>catch</code>	<code>else</code>	<code>namespace</code>	<code>short</code>	<code>_Unpacked</code>
<code>_CC</code>	<code>enum</code>	<code>_Near</code>	<code>signed</code>	<code>unsigned</code>
<code>char</code>	<code>extern</code>	<code>new</code>	<code>sizeof</code>	<code>using</code>
<code>class</code>	<code>_Far</code>	<code>_Noalias</code>	<code>static</code>	<code>virtual</code>
<code>const</code>	<code>float</code>	<code>operator</code>	<code>struct</code>	<code>void</code>
<code>continue</code>	<code>for</code>	<code>_Packed</code>	<code>switch</code>	<code>volatile</code>
<code>__declspec</code>	<code>friend</code>	<code>pragma</code>	<code>template</code>	<code>while</code>
	<code>goto</code>		<code>this</code>	

**WARNING:** `_Packed` and `_Unpacked` are not implemented for all target platforms. See your `README` file for information.

## 3.7 Operators and Their Precedence

C and C++ have a bewildering number of operators and precedence rules to describe the relationships between operators and operands within expressions. However, once you get the hang of operator precedence, associativity, and the use of a precedence table (for example, Table 3.3: *Precedence and Associativity of C++ Operators* on page 21), understanding the structure of C and C++ expressions is not difficult.

*Operators  
“compete” for  
operands*

Think of the operators in an expression as competing for operands. For example, in the expression `a && b < c`, the operators `&&` and `<` are competing for the middle operand `b`. The two operators are not competing for the `a` and `c`; obviously `a` is the left operand of `&&` and `c` is the right operand of `>`.

The question here is whether the expression is equivalent to

`(a && b) < c`

or

`a && (b < c)`

*Using a precedence table* The way to answer this question is to consult a precedence table. Table 3.3: *Precedence and Associativity of C++ Operators* on page 21 shows whether `&&` or `<` has the greater precedence. Table 3.3 shows that `&&` has precedence level 7, while `<` has precedence level 12. Because 12 is greater than 7, `<` “takes precedence over” `&&` when competing for operands.

Thus, the expression `a && b < c` is equivalent to

`a && (b < c)`

not

`(a && b) < c`

That is, the left operand of `&&` is `a`; its right operand is the subexpression `b < c`; the left operand of `<` is `b` and its right operand is `c`.

*Checking associativity* What happens when two competing operators have the same precedence? Check their associativity (they must both have the same associativity).

Consider the expression `a > b < c`. Is it equivalent to `(a > b) < c` or to `a > (b < c)`?

Table 3.3 on page 21 shows that the comparative operators are all at precedence level 12, but also that they are left-associative. This means that when they are competing for the same operand, the one on the left wins.

Thus the expression `a > b < c` is equivalent to `(a > b) < c`, not to `a > (b < c)`.

In general, first use the precedence levels, then break ties based on associativity.

*Precedence of unary operators* As Table 3.3 shows, the prefix unary operators (those that precede their only operand) have lower precedence than the postfix unary operators (those that follow their only operand). Thus, in any competition, the postfix operator wins over a prefix operator (that is, the postfix takes precedence over the prefix).

Consider, for example, the expression `& a ++ (b)`. This is equivalent to `& ((a ++)(b))`; that is, you post-increment `a`, apply the result to `b` (call the referenced function with `b` as parameter), and then dereference the result.

*Operands in expressions* At the bottom of Table 3.3 are the ultimate operands, the constants, variables, or other data items that can appear in expressions. Also, parenthesized (sub)expressions are at the bottom, and operators of any precedence level can be inside the parentheses.

*Exceptions to the C/C++ precedence scheme* Unfortunately, C and C++ have a few exceptions to the usual precedence scheme, as indicated in Table 3.3.

The subexpression that is the left operand of `=`, `op=`, and `? :` cannot contain an operator with precedence level less than 6 (that is, the left operand must have precedence level 6 or greater) unless that operator is found in a parenthesized subexpression.

There are no operators with precedence level 3 or 16, for a technical reason beyond the scope of this discussion.

`sizeof` can have as its operand a `sizeof` subexpression or other prefix-operator subexpression, but not a cast subexpression (unless it is parenthesized). The other prefix operators can have as their operand a subexpression containing any unparenthesized prefix operator.

**Table 3.3** *Precedence and Associativity of C++ Operators*

Prec. Level	Operator	Associativity	Special Notes
1	,	left	
2	<code>=&gt;</code>	none	Cannot have two <code>≥</code> s competing
4	<code>=</code>	right	Precedence(Left Operand) $\geq 6$
4	<code>op=</code>	right	Precedence(Left Operand) $\geq 6$
5	<code>? :</code>	right	Precedence(Left Operand) $\geq 6$ , and Precedence(Middle Operand) $\geq 1$
6	<code>  </code>	left	
7	<code>&amp;&amp;</code>	left	
8	<code> </code>	left	
9	<code>^</code>	left	

**Table 3.3 Precedence and Associativity of C++ Operators (Continued)**

Prec. Level	Operator	Associativity	Special Notes
10	<code>&amp;</code>	left	
11	<code>==</code>	left	
11	<code>!=</code>	left	
12	<code>&lt;</code>	left	
12	<code>&gt;</code>	left	
12	<code>&lt;=</code>	left	
12	<code>&gt;=</code>	left	
13	<code>&gt;&gt;</code>	left	
13	<code>&lt;&lt;</code>	left	
14	<code>+</code>	left	
14	<code>-</code>	left	
15	<code>*</code>	left	
15	<code>/</code>	left	
15	<code>%</code>	left	
17	<code>.*</code>	left	
17	<code>-&gt;*</code>	left	
18	<code>cast_expr</code>	unary prefix	Operand can be any prefix (PL≥18)
19	<code>sizeof</code>	unary prefix	Precedence Level(Operand)≥19
19	<code>+</code>	unary prefix	Operand can be any prefix (PL≥18)
19	<code>-</code>	unary prefix	"
19	<code>*</code>	unary prefix	"
19	<code>&amp;</code>	unary prefix	"
19	<code>!</code>	unary prefix	"
19	<code>~</code>	unary prefix	"
19	<code>++</code>	unary prefix	"
19	<code>--</code>	unary prefix	"
19	<code>delete</code>	unary prefix	Operand can be any prefix (PL≥18)
19	<code>new</code>	unary prefix	"

**Table 3.3 Precedence and Associativity of C++ Operators (Continued)**

Prec. Level	Operator	Associativity	Special Notes
20	<code>++</code>	unary postfix	Operand can be any postfix (PL≥20)
20	<code>--</code>	unary postfix	"
20	<code>(expr)</code>	unary postfix	As in a function call: <code>f(args)</code>
20	<code>[]</code>	unary postfix	"
20	<code>-&gt;</code>	left	
21	<i>constant</i>	operand	
21	<i>variable</i>	operand	
21	<code>(expr)</code>	operand	Parenthesized subexpression
<b>Note:</b> The compound assignment operators are defined in §6.4.1: <i>Assignment</i> on page 79.			

### 3.7.1 Unary Operators

A *unary operator* takes only one operand. Table 3.4 lists the unary operators used in High C++.

**Table 3.4 Unary Operators**

Operator	Description
<code>&amp;</code>	Address operator; yields the memory address of an object
<code>*</code>	Dereference operator; uses the operand as an address and fetches the object at that address
<code>+</code>	Unary plus; no operation
<code>-</code>	Unary minus; negates the operand
<code>~</code>	Bitwise complement
<code>!</code>	Logical negation
<code>++</code>	Used before an operand, pre-increment; used after an operand, post-increment

**Table 3.4 Unary Operators (Continued)**

Operator	Description
--	Used before an operand, pre-decrement; used after an operand, post-decrement
sizeof	The size of an operand in bytes
delete	Deallocate dynamic memory
new	Allocate dynamic memory

### 3.7.2 Binary Operators

A *binary operator* takes two operands. Table 3.5 lists the binary operators used in High C++.

**Table 3.5 Binary Operators**

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder
>>	Shift right
<<	Shift left.
&	Bitwise AND
^	Bitwise exclusive OR
	Bitwise OR
&&	Logical AND
	Logical OR
=	Assignment
=	ORs right operand with left operand and assigns to left operand
^=	Exclusive ORs right operand with left operand; assigns to left operand
&=	Bitwise ANDs left and right operands; assigns to left operand



**Table 3.5 Binary Operators (Continued)**

Operator	Description
<code>&gt;&gt;=</code>	Shifts left operand right by the number of bit positions indicated by the value of the right operand; assigns to the left operand
<code>&lt;&lt;=</code>	Shifts left operand left by the number of bit positions indicated by the value of the right operand; assigns to the left operand
<code>--</code>	Subtracts right operand from left operand; assigns result to left operand
<code>+=</code>	Adds left and right operands; assigns sum to left operand.
<code>/=</code>	Divides left operand by right operand; assigns quotient to left operand
<code>*=</code>	Multiplies left and right operands; assigns result to left operand
<code>&lt;</code>	Less than
<code>&gt;</code>	Greater than
<code>&lt;=</code>	Less or equal
<code>&gt;=</code>	Greater or equal
<code>==</code>	Equal: compare operands for the same value
<code>!=</code>	Not equal: compare operands for different values
<code>.</code>	Select a component of a <b>class</b> , <b>struct</b> , or <b>union</b>
<code>-&gt;</code>	Selects the component of a <b>class</b> , <b>structure</b> , or <b>union</b> denoted by the right operand, indirectly through the address represented by the left operand
<code>::</code>	Scope access and resolution
<code>.*</code>	Dereferences a pointer to a class member, given a class object
<code>-&gt;*</code>	Dereferences a pointer to a class member, given a pointer to a class object
<code>&lt;-</code>	Invokes a High C/C++ iterator

Extension
-----------

### 3.7.3 Ternary Operators

A *ternary operator* takes three operands. High C/C++ has one ternary operator, “**? :**”, used as follows:

```
a ? x : y
```

This expression is equivalent to **b** in the context of this assignment:

```
if (a) b=x; else b=y;
```

For further information about operators, see Chapter 6: *Expressions and Conversions*.

## 3.8 Constants

A *constant* is a value that cannot change during the execution of your program. C++ has several kinds of constants:

- decimal
- floating point
- octal
- hexadecimal
- character

### 3.8.1 Decimal Constants

A decimal constant is a non-zero digit followed by a sequence of zero or more digits, optionally followed by a **long** and/or **unsigned** suffix. Whitespace cannot appear between the digits of an integer constant, or between the digits and the suffix. This is the syntax for a decimal constant:

```
nonzero_digit [digit] . . . [l | L] [u | U]
```

where **l** or **L** implies **long**, and **u** or **U** implies **unsigned**.

*Maximum and minimum values* The maximum and minimum values for an integer are hardware dependent. However, on architectures where **long** means 32 bits, the smallest **signed** integer is -2,147,483,648 and the largest **signed** integer is 2,147,483,647. Similarly, for 32-bit architectures, the smallest **unsigned** integer is 0, and the

largest **unsigned** integer is 4,294,967,295. For more information, consult the **High C/C++ Programmer's Guide** for your target hardware.

If the integer constant has no suffix, its data type depends on the value of the constant. An integer constant with no suffix is assigned the first of these data types that its value will fit in:

```
int
long int
unsigned long int
```

#### Extension

As an extension to C and C++, in High C/C++ you can use one or more underscores between digits of an integer constant.

```
1_200 // This is equivalent to 1200.
```

## 3.8.2 Floating-Point Constants

A floating-point constant is defined as follows:

```
Mantissa [Exponent][float_suffix]
```

where the components have the following meanings:

*Mantissa* is one of the following:

```
digits (If there is no decimal point, an Exponent is required.)
.digits
digits.
digits.digits
```

*Exponent* is:

```
[E|e][+|-] digits
```

*digits* are:

```
[0|1|2|3|4|5|6|7|8|9]...
```

*float\_suffix* is:

```
[f|F][l|L] or [l|L][f|F]
```

The following rules apply to floating-point constants:

- Either the integer part or the fraction part of the constant can be omitted, but not both.
- Either the decimal point or the `e` (or `E`) can be omitted, but not both.
- The exponent is optional, and can have an optional sign preceding it.
- The constant can be followed by the letter `f` or `F`, and/or the letter `l` or `L`.
- If there is no suffix, floating-point constants have the data type `double`.
- If the `f` or `F` suffix is present, the floating-point constant has the data type `float`.
- If the `l` or `L` suffix is present, the floating-point constant has the data type `long double`.

---

### 3.8.3 Octal Constants

An octal constant starts with a `0` (zero):

```
0 octal_digit [octal_digit]...[l|L] [u|U]
```

where `octal_digit` is one of the following ASCII characters:

```
0 1 2 3 4 5 6 7
```

Whitespace cannot appear between the digits of an octal constant or between the digits and the suffixes. The optional suffix `l` or `L` implies `long`. The optional suffix `u` or `U` implies `unsigned`. You can use an octal constant wherever a decimal integer can appear. If an octal constant appears with no suffix, its data type is the first of the following data types that is large enough to contain its value:

```
int
unsigned int
long int
unsigned long int
```

---

### 3.8.4 Hexadecimal Constants

A hexadecimal constant starts with a `0` (zero) followed by `x` or `X`:

```
{0x|0X}{hex_digit}...[l|L][u|U]
```

where *hex\_digit* is one of the following ASCII characters:

0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f

In a hexadecimal constant, the uppercase or lowercase letters **A..F** and **a..f** correspond to the following decimal values:

a	A	10
b	B	11
c	C	12
d	D	13
e	E	14
f	F	15

The optional suffix **l** or **L** implies **long**. The optional suffix **u** or **U** implies **unsigned**. Hexadecimal constants can appear wherever a decimal constant can appear. If a hexadecimal constant appears with no suffix, its data type is the first of the following data types that is large enough to contain its value:

```
int
unsigned int
long int
unsigned long int
```

#### Extension

As an extension to C and C++, you can specify a number in any base from 1 to 16. High C/C++ compilers recognize a second or later **x** following **0x** as a switch to a new base for interpreting the values after the **x**. Here are some examples of using this base conversion:

```
0xfffe           // A normal hexadecimal constant
0x2x1001         // 0x, followed by change to base 2
                  // (binary), followed by binary digits
                  // The value is 910
0x2x1001x20      // Here the binary digits specify a new
                  // base, base 9; the value is 1810,
                  // which is "20" in base 9
```

### 3.8.5 Character Constants

A *character constant* is a sequence of one or more characters enclosed in single quotes. The sequence represents a single character value of type **char** or a multi-character value of type **int**. These are examples:

```
'a'
'AbC'
'12z3'
```

*Multi-character constants* The value of a multi-character constant is implementation dependent. If you plan to move your C++ program to many different machines, you should avoid using multi-character constants, and use strings instead.

**C ≠ C++**

In C++, single-character constants have data type **char**, but in ANSI Standard C they have type **int**. If you write **sizeof(char\_const)**, where **char\_const** is any single-character constant, in C++ you get the value 1 (one), while in ANSI Standard C you get **sizeof(int)**.

*Wide characters* A character constant whose integral value exceeds 256 (0377 octal, 0xFF hexadecimal) is undefined, unless it is immediately preceded by the letter **L**. Here is an example of the use of the **L** character as a prefix:

```
L'ab'
```

This is interpreted as a “wide” character, and has the data type **wchar\_t**. Wide characters are used for international character sets where the value of a character does not fit into a single byte.

### 3.8.5.1 Escape Sequences

Within a character constant or string, a backslash character (**\**) begins an escape sequence. An *escape sequence* means that the backslash is to be combined with the character or characters that follow it to form a visual representation of a certain non-printing graphic character, or a two-character representation of the characters **'**, **"**, **?**, and **\**. Table 3.6 shows the escape sequences defined in High C++.

**Table 3.6** *Escape Sequences*

Escape Sequence	Description	Value
'\n'	ASCII <b>LF</b>	Newline
'\t'	ASCII <b>HT</b>	Horizontal tab
'\v'	ASCII <b>VT</b>	Vertical tab
'\b'	ASCII <b>BS</b>	Backspace
'\r'	ASCII <b>CR</b>	Carriage return

**Table 3.6** *Escape Sequences (Continued)*

Escape Sequence	Description	Value
'\f'	ASCII <b>FF</b>	Form feed
'\a'	ASCII <b>BEL</b>	Bell
'\\'	\	Backslash
'\?'	?	Question mark
'\''	'	Single quote
'\"'	"	Double quote
'\o...'	<i>o</i> , <i>oo</i> , or <i>ooo</i>	Octal number: one to three octal digits
'\xh...'	<i>h</i> ...	Hexadecimal number: any number of hexadecimal digits

**Note:** An escape sequence of the type `\o...` must contain at least one and no more than three octal digits.

In an escape sequence of the type `\xh...`, the `\x` can be followed by any number of hexadecimal digits. The first non-hexadecimal character defines the end of the sequence.

### 3.8.6 Strings

A *string* is a sequence of characters enclosed in double quotes. Escape sequences (see §3.8.5.1: *Escape Sequences* on page 30) are valid in strings as well as in character constants. To represent the character `"` in a string, you must use the escape sequence `\"`.

Strings have the data type `char[ ]` (that is, array of `char`), unless the string is immediately preceded by the letter `L`. A string immediately preceded by `L` has the data type `wchar_t[ ]` (that is, array of `wchar_t`), and represents an array of wide characters.

If two strings are adjacent in C++, they are concatenated. For example:

```
cout << "How are " "you?"
```

The adjacent strings "How are " and "you?" are equivalent to the single string "How are you?". All strings in C++ are null-terminated.

## 3.9 Trigraphs

A *trigraph* is a sequence of three characters that represents another character. If your computer system does not support the full ASCII character set, or if your system assigns foreign-language symbols to certain character values in the ASCII character set, you might have to use trigraphs to represent particular ASCII characters. The meanings of trigraphs hold everywhere, even in strings or constants.

Table 3.7 lists the trigraphs defined in C++.

**Table 3.7** *Trigraphs in C++*

Trigraph	Meaning	Trigraph	Meaning
??=	#	??(	[
??/	\	??)	]
??'	^	??!	

To actually use the characters ?? in a string or character constant, you must use the escape sequence \?:

```
printf("What?\?!"); // Prints: What??!
```

whereas:

```
printf("What??!"); // Prints: What|
```



# 4

## Preprocessing C and C++ Program Text

---

When you compile a C or C++ program, the compiler first preprocesses the program text. Preprocessing does the following:

- splices lines
- removes comments
- executes preprocessor directives
- expands macro references
- replaces escape sequences in literals with their equivalent character representations
- concatenates adjacent string literals

**C ≠ C++**

Except for the treatment of `/// style comments, preprocessing in C++ is the same as preprocessing in ANSI Standard C.`

---

### 4.1 Line Splicing

C or C++ source code is usually divided into a sequence of separate lines of text. The C and C++ languages do not require statements to appear on separate lines, nor do they impose a limit on the length of a source line. A single token, such as an identifier, can actually span several source lines, by means of *line splicing*.

*Glue source lines  
together with a  
backslash character*

Line splicing means taking a source line and the line that immediately follows it, and “glueing” them together, as though there were no newline character separating them. Each source line whose last character is backslash (`\`) is concatenated with the following line after deleting the backslash and the newline character, as shown in the following example.

```

i\
am\
all\
one\
name++;
iamallonenam++; // Equivalent to the spliced line

```

In effect, `\` followed by a newline character means to the preprocessor “act like we are not here: concatenate the former with the following”. Any token can be spliced across line boundaries with the backslash character, including comments and strings.

*You can splice* You should be careful, though, when using line splicing with C++ end-of-line  
*//-style comments* comments. For example:

```

#define big_macro(i,j) {i++; // Comment \
                        j++; // Another comment \
                        }

main(){
    int i,j;
    big_macro(i,j);
}

```

This example shows a macro definition using `#define` (see §4.2: *Preprocessor Directives* on page 35), where the definition is continued across source-line boundaries using line splicing. C++ end-of-line comments appear at the end of the first two source lines. The remainder of the macro, following the first end-of-line comment, is treated as part of the comment. The preceding example expands to the following text after preprocessing:

```

main(){
    int i,j;
    {i++;          ;
    }
}

```

This is not valid C++ syntax, and will generate error messages at compile time. To get what was probably the intended effect in this example, you could change the comments to the `/* */` form.

## 4.2 Preprocessor Directives

*Control lines* A line of source-program text that begins with a preprocessor directive is a *control line*.

The *preprocessor directives* are listed in Table 4.1:

**Table 4.1 Preprocessor Directives in High C/C++**

<code>#define</code>	<code>#endif</code>	<code>#ifdef</code>	<code>#line</code>
<code>#elif</code>	<code>#error</code>	<code>#ifndef</code>	<code>#pragma</code>
<code>#else</code>	<code>#if</code>	<code>#include</code>	<code>#undef</code>

### Extension

The following High C/C++ extensions are also preprocessor directives:

```
#c_include
#print
```

A control line can have whitespace characters preceding the `#` character that begins a preprocessor directive. The preprocessor directive name must be one of the names listed above, or empty.

*Null directive* If nothing but whitespace follows the `#` character on the source line, it is the *null directive*. A null directive is ignored, and has no effect.

### 4.2.1 `#include` and `#c_include`

As in C, the `#include` directive imports library functions and files from outside the current program. When you use the `#include` directive, you must follow it with a file specification. The `#include` places the source text from the named file at the point of the `#include`.

High C/C++ provides two forms of the `#include` and `#c_include` directives:

```
#include <filename>    or    #c_include <filename>
#include "pathname"    or    #c_include "pathname"
```

These rules apply to the **#include** and **#c\_include** preprocessor directives:

- The newline character cannot appear in either *filename* or *pathname*.
- The > character cannot appear in *filename*.
- The " character cannot appear in *pathname*.
- Whitespace is not allowed in either form between the <> or " " delimiters.
- If any of the characters ' , \, or " or the sequences /\* or //, appear between the delimiters, the interpretation of the token string between the delimiters is undefined.

In the <*filename*> form, the token string *filename* is not required to specify a complete path for the location of the file you want to include. The compiler will look for this file in the High C/C++ include directory, if any, followed by the host operating system's include directory, if any.

See the **High C/C++ Programmer's Guide** for information about the search order and locations in which the compiler looks for include files.

*Includes can be  
nested*

A file specified in either <*filename*> or "*pathname*" can contain further **#include** directives. Consult the **High C/C++ Programmer's Guide** for your host operating system for limits on the number of include files that can be so nested.

#### Extension

As a language extension, you can use the directive **#c\_include** — conditional file inclusion. **#c\_include** works exactly like **#include**, except that if the file to be included has already been included once in the source file, it will not be included again. **#c\_include** is therefore a *conditional* file inclusion facility that helps you avoid duplicate declarations. Also, using **#c\_include** avoids file I/O and compilation overhead because the included source file is processed only once.

## 4.2.2 #define and #undef

C and C++ allow you to define macro code sequences and symbolic constants. The **#define** preprocessor directive associates a symbolic name with a code sequence or constant. The **#define** directive has the following form:

```
#define macro_name [ (parm [ ,parm ] . . . ) ] token_string
```

Following any needed *parm* substitutions, *token\_string* is substituted for occurrences of *macro\_name* and its associated optional parameter list in the

source program. This substitution can occur anywhere after the **#define** directive, unless (or until) an **#undef *macro\_name*** directive is encountered.

*Macro names are usually uppercase* *macro\_name* can be any valid identifier (see §3.5: *Identifiers* on page 18) and it is followed by an optional parameter list; each *parm* must be a valid identifier. By convention, *macro\_name* is often in uppercase, although this is not required.

Each optional parameter must be a valid identifier, with parameters separated by commas. *token\_string* can be whitespace, or any valid sequence of C or C++ tokens. You can splice *token\_string* across line boundaries using the backslash (\) character (see §4.1: *Line Splicing* on page 33); that is, *token\_string* must be contained on one line after splicing. For example:

```
#define FAR
#define MAX(x,y) ((x) > (y) ? (x) : (y))
#define MAX_BUFFERS 30
#define RPAREN )
#define LPAREN (
#define ADD(x,y) x +\
                y
main() {
    FAR int *ptr;
    int i = 40;
    int j = 50;
    i = LPAREN ADD(i,j) RPAREN;
    if ( MAX_BUFFERS < MAX(i,j) )
        i = MAX_BUFFERS;
}
```

After preprocessing, this program would look something like the following:

```
main() {
    int *ptr;
    int i = 40;
    int i = 50;
    i = ( i + j );
    if ( 30 < ((i) > (j) ? (i) : (j)) )
        i = 30;
}
```

*End a macro definition with* The **#undef** directive terminates the association between *macro\_name* and *token\_string*. This is the form of an **#undef** directive:

```
#undef
      #undef macro_name
```

---

### 4.2.3 Conditional Directives

Preprocessing provides a way to conditionally compile your programs. You can include or omit sections of code based on whether a particular *macro\_name* is defined, or whether a constant expression has a non-zero value. A common use for conditional compilation is to include or omit printing information to help in debugging your program. Conditional compilation directives have the following form:

```
if_line
    source_text
[ elif_line
    source_text ] ...
[ else_line
    source_text ]
#endif
```

where the components have the following meanings:

*if\_line* can be one of the following:

```
#if constant_expression
#ifdef identifier
#ifndef identifier
```

*source\_text* is zero or more valid C or C++ tokens.

*elif\_line* is:

```
#elif constant_expression
```

*else\_line* is:

```
#else
```

*Nested conditional directives* You can nest conditional compilation constructs. There is no particular limit on the number of levels of nesting.

#### 4.2.3.1 `#if`

When the preprocessor encounters `#if`, the *constant\_expression* is evaluated. If the result is a non-zero value, the source text following the *if\_line* is included in the compilation. If the *constant\_expression* evaluates to 0 (zero), the source text following the `#if` is ignored, up to the matching `#endif`, `#else`, or `#elif`.

#### 4.2.3.2 `#ifdef` and `#ifndef`

An `#ifdef` directive is similar to an `#if` directive, except that instead of evaluating a constant expression, the preprocessor checks to see if *identifier* has been defined, by means of a `#define` directive or a compiler command-line definition. If *identifier* is defined, the source text following the `#ifdef` directive is included in the compilation. Otherwise, the source text is ignored.

`#ifndef` is the opposite of `#ifdef`: source text following an `#ifndef` directive is included in the compilation only if *identifier* has not been defined.

See the **High C/C++ Programmer's Guide** for information about the compiler command line.

#### 4.2.3.3 `#elif` and `#else`

An `#elif` directive is valid only within a surrounding `#if` / `#endif` pair. `#elif` is processed only if the matching preceding `#if` or `#elif` directive evaluates to 0 (zero), or the `#ifdef` or `#ifndef` failed.

If the *constant\_expression* following `#elif` evaluates to a non-zero value, the source text following `#elif` is included in the program, and all text following a succeeding `#elif` or `#else` is skipped until the corresponding `#endif` is encountered. In this context, a *constant\_expression* cannot contain a `sizeof` operator, a type cast, or an enumeration constant.

For a definition of *constant\_expression*, see §6.3: *Constant Expressions* on page 78.

If all matching preceding `#if`, `#ifdef`, `#ifndef`, and `#elif` directives evaluate to 0 (zero), the source text following the `#else` directive is included in the compilation.

#### 4.2.3.4 Unary Operator **defined**

Each *constant\_expression* can contain a unary operator **defined**. **defined** can be used in either of the following forms:

```
defined(x)  
defined x
```

**defined**(*x*) (or **defined** *x*) evaluates to 1 (one) if *x* is a defined macro name; otherwise it evaluates to 0 (zero). Thus, **#if defined**(*x*) is equivalent to **#ifdef** *x*; however, unlike **#ifdef**, **defined**(*x*) can be combined in a complex expression such as:

```
#if defined(x) && !defined(y)
```

You cannot undefine or redefine the identifier **defined**.

---

#### 4.2.4 **#line**

For programs such as translators that preprocess C++ as input and produce C/C++ output, C++ provides the **#line** preprocessor directive. This is the form of the **#line** directive:

```
#line compile_time_constant [ "filename" ]
```

where *compile\_time\_constant* is a valid integer constant, and the optional *filename* string is a valid file name for the host operating system. If you use the **#line** directive, the macro constant **\_\_LINE\_\_** is set to the *compile\_time\_constant* value as the next line is processed.

If you use the optional *filename* string, **\_\_FILE\_\_** is set to equal the value of *filename*. *compile\_time\_constant* can be a *macro\_name* or can contain a *macro\_name*. Macro replacement is completed before the **#line** directive is processed.

---

#### 4.2.5 **#error** and **#print**

The preprocessor directive **#error** has the following form:

```
#error token_string
```



where *token\_string* is any sequence of printable characters (on one line, after line splicing). **#error** causes *token\_string* to be displayed as a diagnostic message during compilation.

Extension

The **#print** directive is a language extension. It works the same way as **#error**, but is not treated as a compile-time error that terminates compilation. **#print** provides a convenient way of printing informational messages.

---

#### 4.2.6 #pragma

The **#pragma** directive causes an implementation-dependent action. You can use pragmas to do such things as:

- control compiler warning levels and messages
- conditionally include source files
- provide an interface between High C or High C++ and other programming languages

A **#pragma** directive has the following form:

```
#pragma Pragma_name [ ( Pragma_parameters ) ] [ ; ]
```

where *Pragma\_parameters* is a list of constant expressions separated by commas.

*Parameters vary  
from pragma to  
pragma*

The number and types of the expressions depend on the particular *Pragma\_name*. If *Pragma\_name* is a valid High C/C++ pragma name, the rest of the source text following the pragma name must be correct according to the definition of that pragma in High C/C++. For example, the *Calling\_convention* pragma has three valid forms:

```
#pragma Calling_convention( )
#pragma Calling_convention(Attributes)
#pragma Calling_convention(Attributes, _DEFAULT)
```

Where *Attributes* is a bit pattern. The following is not correct:

```
#pragma Calling_convention(1, 2, 3) // Not valid
```

Although *Calling\_convention* is a valid High C/C++ pragma, the values inside the parentheses are not valid for this pragma, and this usage draws an error message. If you use a pragma name that is not a valid High C/C++ pragma name, you receive a warning message.

See the **High C/C++ Programmer’s Guide** for information about High C/C++ pragmas.

### 4.3 Predefined Macro Names

Some macro names are defined for you. These names cannot be redefined or undefined. Table 4.2 lists the predefined macro names.

**Table 4.2** *Predefined Macro Names*

	<code>__LINE__</code>	A decimal constant: the current C++ source-file line number
	<code>__FILE__</code>	A string: the name of the source file being compiled
	<code>__DATE__</code>	A string: the date of compilation in the form <code>mmm dd yyyy</code> (for day values 10 or greater; for days 1 through 9, the date has the form <code>mmm d yyyy</code> )
	<code>__TIME__</code>	A string: the time of compilation, in the form <code>hh:mm:ss</code>
	<code>__cplusplus</code>	Defined when compiling a C++ program; the same as <code>#define __cplusplus 1</code>
Extension	<code>__CPLUSPLUS__</code>	The same as <code>__cplusplus</code> ; a High C++ extension
	<code>__STDC__</code>	Defined when compiling C programs, or programs that are a mixture of ANSI Standard C and C++
Extension	<code>__HIGHC__</code>	Predefined to be 1 (one) for all implementations of High C unless compiling in ANSI Standard C mode; can be used to exclude code containing High C/C++ extensions, for example

### 4.4 Concatenation

Operators `#` and `##` are permitted within the replacement token sequence `R` of a definition of a parameterized macro. Within `R` these operators have special meanings. These meanings change the way token substitution is performed for parameter names that appear as arguments to `#` and `##`.

Macros are usually expanded before substitution

Normally, when a parameter name `N` appears in `R`, `N` is replaced by the token sequence that was given as the argument for `N` at the macro invocation, after macro expansion has first been performed on that token sequence. That is,

macro expansion is first applied to the argument token sequence, then substitution of the expanded text occurs wherever **N** appears in **R**.

*Tokens can be glued together before macro expansion*

If **#** is followed by a macro parameter name **N** within **R**, both are replaced by a single character-string token that contains the spelling of the unexpanded argument token sequence for **N** — that is, before macro expansion has been applied to the argument. The spelling is defined as the concatenation of the texts of each of the individual tokens, with a single space inserted between any two tokens that have any amount of whitespace (comments, spaces, tabs, and so forth) between them, and where any occurrence of **"** or **\** is prefixed by **\** (to ensure that when compiled the string produces **"** or **\** correctly). See §4.4.1: *Uses of Operators # and ##* for examples.

If a macro parameter name **N** is immediately preceded and/or followed by **##**, the parameter is replaced by the unexpanded argument token sequence. Before the replacement text is re-examined for further macro expansion, each instance of **##** is deleted, and the two tokens preceding and following the **##** are replaced by a single token whose text contains the concatenation of the texts of the two tokens. The resulting concatenation must be a valid token, otherwise the result is undefined. If there are no such two tokens, the result is undefined. The resulting single token is considered for expansion when the replacement text is re-examined for further macro expansion.

**##** cannot appear as the first or last token of a replacement token sequence.

---

#### 4.4.1 Uses of Operators **#** and **##**

A typical use of **#** would be for printing the text of a variable or expression in a program:

```
#define str(y) #y
#define debug(x) \
    printf("The value of " str(x) " is %d", x)
```

If the **int** variable **xyz** has the value 3, the invocation **debug(xyz)** prints:

```
The value of xyz is 3
```

and **debug(xyz+ /\*comment\*/ 4)** prints:

```
The value of xyz+ 4 is 7
```

Notice the single space inserted between `+` and `4`, corresponding to the whitespace between `+` and `4` in the argument to `debug`.

```
debug('a')
```

prints:

```
The value of 'a' is 97
```

(assuming an ASCII target) because `'a'` is converted to the C/C++ string `"'a'"`.

A typical use of `##` is for the construction of an identifier from a distinct identifier prefix and suffix:

```
#define Glue(x,y) x ## y
#define OS_CPU(Prefix) char *Glue(Prefix,_OS), \
                          *Glue(Prefix,_CPU)
```

Then:

```
OS_CPU(VAX)
OS_CPU(SUN)
```

declares four `char *` variables: `VAX_OS`, `VAX_CPU`, `SUN_OS`, and `SUN_CPU` (assuming `VAX` and `SUN` are not parameterless macros that expand to something else).

To illustrate the substitution of non-expanded argument token sequences, consider the following:

```
#define Str(x) #x
#define Str2(x) Str(x)
#define abc def
```

Then:

```
Str(abc)    // Expands to "abc"
Str2(abc)   // Expands to "def"
```

First `abc` is expanded to `def`, then `def` is substituted in the replacement text for `Str2`, producing `Str2(def)`; then, `def` is converted to a string. Another example:

```
#define Glue(x,y) x ## y
#define Glue2(x,y) Glue(x,y)
```

Then:

```
Glue(abc,abc)    // Expands to abcabc
Glue2(abc,abc)   // Expands to defdef
```

Finally, some error situations:

```
#define Bad1(x) ## x ##    // Both ##s are invalid.
#define Bad2(x) #y         // y is not a formal
                           // parameter.
#define Glue(x,y) x ## y
...
Glue(,123)           // Invalid: There are no tokens
                     // to the left of ##SQ.
```



# Declarations and Definitions

---

- Declarations* A *declaration* associates a name with a specific set of attributes. Among those attributes are location in storage, data type, size, and storage category. All names must be declared in C++ programs before they are used.
- Definitions* If a declaration allocates storage, assigns a value, or defines a function body, the declaration is also a *definition*.
- Variables* A *variable* is a declared name whose value can change during the execution of your program. For example:

```
int i; // This is both a declaration and a
      // definition because it reserves space
      // for the variable i.
char myfunc( char *s) { return *s; }
      // The preceding is also a definition, because
      // it defines a function body.
extern atan(float); // Declaration only
struct astruct;    // Declaration only
```

In this example, `i` is declared as an integer. The declaration gives the data type and size for `i`, and causes storage to be reserved for `i`. See §5.4: *Data Types* on page 52 for more information about type declarations.

The declaration of `myfunc()` defines a function that returns a single character, after dereferencing the input parameter. This declaration is a definition because it defines a function body. This is the body of the function:

```
{ return *s; }
```

The declaration of the function `atan()` is not a definition because no function body is defined. The `extern` keyword informs the compiler that the body of the function is defined elsewhere. Similarly, `struct astruct` simply declares that `astruct` is a structure, without reserving any storage or defining what `astruct` looks like.

## 5.1 Declarations

---

A *declaration* is one of the following:

```
[ decl_specifiers ] [ init_declarator
                        [ , init_declarator ] ... ]

function_definition
linkage_specification
```

See §5.6: *Linkage Specifications* for a discussion of *linkage\_specification*.

*decl\_specifiers* are:

```
[ storage_class_specifier
  type_specifier
  inline
  virtual
  template
  friend
  typedef
  ... ]
```

*init\_declarator* is:

```
declarator [ initializer ]
```

See §5.7: *Initialization* for a discussion of *initializer*.

*declarator* is one of the following:

```
declared_name
ptr_operator declarator
declarator ( argument_decl_list ) [ cv_qualifier_list ]
declarator [ [ constant_expression ] ] ( declarator )
```

*declared\_name* is one of the following:

```
name
~identifier
[ templat_invocation ] [ identifier :: ] ... name
```



*name* is one of the following:

*identifier*  
*operator\_function\_id*  
*conversion\_function\_id*

*ptr\_operator* is one of the following:

[ *templat\_invocation* ] [ *identifier ::* ] ...  
                                   \* [ *cv\_qualifier\_list* ]  
                                   & [ *cv\_qualifier\_list* ]

*cv\_qualifier\_list* is:

{ *const* | *volatile* } ...

*Names are identifiers* A *name* is usually just an *identifier*. The form of an *operator\_function\_id* is shown in §12.3: *Operator Functions* on page 198. The form of a *conversion\_function\_id* is shown in §11.3: *User-Defined Conversions* on page 179.

---

## 5.2 Blocks and Name Scoping

A *block* is a collection of declarations and statements enclosed by curly braces { }. A block can appear anywhere a statement can appear; see Chapter 7: *Statements* for more information about statements. Objects declared within a block are accessible only within the block, unless they are declared using **extern**, or unless you use the scope resolution operator :: (see §5.4.2: *Scope Access* on page 55).

*Functions are blocks* A function is a special kind of block that can be called, might take parameters, and might return a value.

*Blocks can be nested* An *inner block* is a block that is enclosed by another block. If you declare a name in an inner block, and you have declared the same name in an enclosing block, you will be unable to access the name in the enclosing block from within the inner block, unless you use the scope resolution operator ::. For example:

```

#include <iostream.h>
float global = 2.75; // Declaration in file scope
main(int argc, char *argv[]) {
    char global; // This declaration is inside block
                  // main. Now you cannot access the
                  // global at file scope from within
                  // main unless you use ::global.

    global = 'a';
    {
        int global = 3;
        cout << global; // Prints "3"
    }
    cout << global;      // Prints "a"
    cout << ::global     // Prints "2.75"
}

```

*Scope of a name* The *scope* of a name is the range of source program lines over which the name is known and accessible. The scope of a given name extends from the line where the name is declared to the end of the block. You can declare names outside any block.

*File scope* Names declared outside any block are said to have *file scope* because they can be used from any block in the source file and anywhere at the file-scope level after they are declared.

*Uniqueness of names* Names that are declared at the same scope level must be unique, unless they are overloaded. See Chapter 12: *Name Overloading and Operator Functions in C++* for more information about name overloading.

---

## 5.3 Storage Categories

When you declare an identifier, you determine the storage to be assigned to the identifier. All variables and functions in C++ programs are assigned a storage category. Categories of storage in C++ are **auto**, **extern**, **register**, and **static**.

---

### 5.3.1 **auto** Storage

Data declared inside a block or routine is automatic. See §5.2: *Blocks and Name Scoping* on page 49 and Chapter 8: *Functions* for more information about blocks and routines. The storage is allocated each time a block or routine is entered. Values in automatic data are lost when execution of the block or routine terminates. You cannot use the **auto** keyword except within a block or routine.

---

### 5.3.2 **extern** Storage

External storage is storage that is allocated in some other source file or library. When you declare a data item **extern**, you are saying not to allocate storage for the data item. For a function, **extern** implies that the function body may be defined in another source file, and the function identifier can be used to call the function from any source file that is a part of the program. You cannot use **auto** or **register** with **extern** in a declaration.

---

### 5.3.3 **register** Storage

When you use **register**, you declare the data item as **auto**, and request that the data item be allocated to a processor register if possible. See the **High C/C++ Programmer's Guide** for more information about the treatment of **register** data items. Generally, the compiler is better than you are at determining which data items need to be in registers. A small change in your program can radically alter the benefits of placing a particular item in a register.

---

### 5.3.4 **static** Storage

Once allocated, this storage is always present. This category of storage is frequently used for data that must be accessible to all functions in a source file, or data that must retain its value in each invocation of a recursive routine. When you declare a **static** function, you must define the body of the function in the same source file. You cannot call a **static** function by name from another source file.

### 5.3.5 Default Storage Categories

All variables and functions in C and C++ programs are assigned a storage category. By default, a variable declared outside any routine or block is statically allocated, but operations in other source files can access the variable, unless it is preceded by the **static** keyword. If you declare a variable inside a routine or block, then by default the variable is **auto**.

You can use only one of the storage-category keywords in a declaration. These keywords cannot be repeated in a single declaration. For example:

```
static static int i;    // This line is not valid,
register auto char c;  // nor is this line.
```

Formally, a *storage\_class\_specifier* is one of the following:

```
auto
register
static
extern
```

## 5.4 Data Types

Each data item in C or C++ has a *type*, which determines how the item can be used and what operations on it are valid. A declaration (see §5.1: *Declarations* on page 48) determines the type, and the type comes from two places: the *declarator* and the *type\_specifier* names (if any) that are given in the declaration.

*Declarators* Basically, a *declarator* supplies type information when you use any of the following forms (see §5.1: *Declarations* on page 48):

```
ptr_operator declarator
declarator ( argument_decl_list ) [ cv_qualifier_list ]
declarator [ [ constant_expression ] ]
```

*Examples of declarations* The first form specifies a pointer or reference type; the second, a function type; and the last, an array type. See §5.4.5.1: *Pointers* on page 60, §5.4.5.2: *References* on page 60, §5.4.5.3: *Arrays* on page 61, and Chapter 8: *Functions* on page 111 for how each declarator specifies a type.

Here are some examples of declarations:

```
int a
char *b
float *c[3]
int d[3][5]
struct s (*e)[3]
f(char)
(*g(double))[4]
int *&h
```

For `int a`, the declarator gives no type; there is just a *declared\_name* `a`. The type is just that supplied by *type\_specifier* `int`, so the type of `a` is `int`.

For `char *b`, the declarator gives the type “pointer to”. Pointer to what? To the type supplied by the *type\_specifier*, namely `char`. So `b` is of type pointer-to-`char`.

For `float *c[3]`, the declarator gives the type array-of-three-pointers-to, and the *type\_specifier* tells what is pointed to. So the type of `c` is array-of-three-pointers-to-`float`.

Briefly:

- `d` is an array-of-three-arrays-of-five-`ints`.
- `e` is a pointer-to-array-of-three-`struct s`.
- `f` is a function taking a `char` and returning `int`. (If no type is given in the *type\_specifiers* for a function, `int` is assumed; see Chapter 8: *Functions*.)
- `g` is a function taking a `double` and returning a pointer-to-array-of-four-`ints`. (Once again, the `int` is implied.)
- `h` is a reference-to-pointer-to-`int`.

In each of the preceding bullets, the final data type in the description (`int`, `struct s`, `int`, `int`, `int`) is contributed by *type\_specifier*.

## 5.4.1 Type Specifiers

A *type\_specifier* is one of the following:

**C++ Only**

```
[::][identifier ::] ... identifier
class_definition
enum_definition
reference_to_previous_class_or_enum_definition
type_qualifier
long
short
unsigned
signed
char
int
void
float
double
```

*reference\_to\_previous\_class\_or\_enum\_definition* is:

```
{class | struct | union | enum} [identifier ::] identifier
```

A *type\_specifier* of the form **class** *X*, **struct** *X*, or **union** *X* is a reference to a previous *class\_definition*. A *type\_specifier* of the form **enum** *X* is a reference to a previous *enum\_definition*. You can also define classes and **enums** when writing a *type\_specifier*; these definitions are covered in §5.4.4: *Type enum* on page 58 and §5.4.5.4: *Structures* on page 62, respectively.

The simplest form of *type\_specifier* is just *identifier*. The *identifier* refers to a previously declared type. For example:

```
typedef float T;
T x;    // T is the identifier.  x is now a float.
```

## 5.4.2 Scope Access

### C++ Only

Scope resolution  
operator `::`

You use the scope resolution operator `::` to access names that are not readily visible in the current scope. For example:

```
struct s {
    typedef float T;
};
s::T x;           // Use :: to get T within s.
```

You can also use the scope resolution operator `::` in a reference to a previous **class** or **enum** definition, although most of the time this is unnecessary. For example:

```
struct s {
    struct t { int z; };
};
struct s::t x;      // x is of type s::t.
s::t y;             // This works as well.
```

The only time you need the prefixing **struct/class/union/enum** keyword is when the **class** or **enum** name is hidden by a “competing” declaration. For example:

```
struct s {
    struct t { int z; };
    int t;           // This hides the name t
};                  // of the struct.
struct s::t x;      // x is of type struct s::t.
s::t y;             // This no longer works, and
                    // draws a syntax error message.
```

In general, however, you almost never have to use the *reference\_to\_previous* form of *type\_specifier*; you can just use the *identifier* instead.

### C ≠ C++

**Note:** In ANSI Standard C, as opposed to C++, you must *always* use the *reference\_to\_previous* form. C does not introduce **class** and **enum** names as type names; C++ does.

### 5.4.3 Fundamental Types

C and C++ have fundamental predefined data types, listed in Table 5.1. The “size” refers to the amount of space taken up by data of a given type.

*Arithmetic types:* **int**, **char**, and enumerated types (see §5.4.4: *Type enum* on page 58) are referred to as *integral types*. **float** and **double** types are *floating types*. Integral and floating types are *arithmetic types*.

**Table 5.1 C and C++ Predefined Data Types**

Type Name	Description
<b>char</b>	A single character; signedness is system specific
<b>unsigned char</b>	A single unsigned character
<b>signed char</b>	A single signed character
<b>int</b>	A signed integer; typically four bytes on a 32-bit machine, two bytes on a 16-bit machine
<b>short int</b>	A <b>signed</b> integer; the same size as an <b>int</b> , or smaller; typically two bytes
<b>long int</b>	A <b>signed</b> integer; the same size as an <b>int</b> , or larger; typically four bytes
<b>long long int</b>	A <b>signed</b> integer; the same size as a <b>long int</b> , or larger; typically eight bytes on machines that support such
<b>float</b>	A floating-point number, which may have a fractional part
<b>double</b>	A double-precision floating-point number; the same size as a <b>float</b> , or larger
<b>long double</b>	A <b>long</b> double-precision floating-point number; the same size as a <b>double</b> , or larger
<b>void</b>	An empty set of values

**Extension**

If you do not specify a type in a declaration, the type defaults to **signed int**.

*Signed versus unsigned types* For **int** and **char** types, you can specify **signed** or **unsigned** explicitly in the declaration. Whether the type **char** is signed or unsigned depends on the particular C or C++ compiler you are using. **char** defaults to unsigned in High C/C++. You can change the default for **char** with compiler toggles. See the **High C/C++ Programmer’s Guide** for more information.



You can specify the keywords **long**, **short**, **unsigned**, **signed**, **int**, and **double** in any order. For example:

```
int unsigned short const x; // These two declara-
const short unsigned int x; // tions are the same.
```

You must not specify conflicting type information in a declaration. For example:

```
long short int i; // Invalid
```

*Size of data types* The actual size in bits or bytes of a given type can vary from one computer system to another. Use the **sizeof** operator to determine the size in bytes of a type or data item on your system. See §6.4.10.9: *The sizeof Operator* on page 90 for more information about **sizeof**.

*Minimum size range* Some types have a guaranteed minimum size. A **char** must be at least eight bits, a **short int** must be at least 16 bits, and a **long int** must be at least 32 bits. In almost all High C/C++ implementations, these sizes are in fact exactly the sizes of those types. A **char** can hold a character of the target machine's character set. Given these requirements, Table 5.2 shows the minimum range of values for these types.

**Table 5.2 Minimum Ranges for Integers**

Type Name	Minimum Range
<b>char</b>	0 to 127
<b>short int</b>	-32,768 to 32,767
<b>long int</b>	-2,147,483,647 to 2,147,483,646
<b>long long int</b>	-4,294,967,296 to 4,294,967,295

**Extension**

**long long int** types are supported only on some systems. Where **long long int** types are supported, the range is typically  $-2^{63}$  to  $2^{63}-1$ ; the range of an **unsigned long long int** is 0 to  $2^{64}-1$ . Where **long long ints** are not supported, the compiler issues a warning and treats such a type as a **long int**.

### 5.4.4 Type **enum**

Use an *enum\_definition* to define an enumeration type. For example:

```
enum color { red, green, blue, violet }
```

In this example, a distinct type, **color**, is defined. **color** is an integral type, and a variable declared of type **color** has **sizeof(char)**. The members of the enumerated type, such as **red**, **green**, **blue**, and **violet**, are called *enumerators*. Each enumerator name must be distinct from any other identifiers in the same scope. By default, the first enumerator has the value 0 (zero), and each successive enumerator's value is one more than the previous enumerator. In this example, **green** has the value 1 (one), **blue** has the value 2, and so on.

---

**Note:** You can use a compiler switch to override the size of **enum** types; see the **High C/C++ Programmer's Guide** for more information.

---

*Initial values for  
enumerators*

You can specify an initial value for an enumerator. The value must be an integer, or be promotable to an integral type. Enumerators within an enumerated type are not required to have distinct values.

This is the form for an *enum\_definition*:

```
enum [ identifier ]  
      { [ enumerator [ , enumerator ] . . . [ , ] ] }
```

where an *enumerator* is defined as:

```
identifier [= constant_expression]
```

**C ≠ C++**

If an *identifier* appears after the **enum** keyword, that name is declared as a type name (C++ only), and can be used as a subsequent *type\_specifier*. Omitting the *identifier* is done primarily in **typedef** declarations. See §5.4.7: *Defining Types with typedef* on page 69 for more information about **typedef**. For example:

```
main(){
    typedef enum { red,
                  green = 3*2,
                  blue,
                  violet = 6,
                  puce } tname;
    // Identifier ^ is omitted, but you can
    // use tname to refer to the type:
    tname mycolor;
    mycolor = blue;
}
```

In this slight modification of `color`, `green` is initialized to the value `3*2` or `6`. This makes the value of `blue` equal to `7`; but then `violet` is initialized to `6`, so `puce` also gets the value `7`.

An *enum\_definition* can appear all by itself; notice the added semicolon to complete the declaration:

```
enum color { red, green, blue, violet };
```

This works because the declaration syntax (see §5.1: *Declarations* on page 48) allows *decl\_specifiers* without any declarator. Here is just such a place: the *enum\_definition* is a *decl\_specifier*, and no declarator is supplied.

#### Extension

High C/C++ allows you to specify `long` and `short` enumeration types. This is an extension to the C and C++ languages.

### 5.4.5 Derived Types

A *derived type* is based on one of the fundamental types, such as `int`, `double`, `float`, `char`, or `void`. Derived types include pointers, references to other types, arrays of other types, class types, function types, structures, and unions. Do not confuse a derived type with the notion of a “derived class” (see §9.6: *Derived and Base Classes* on page 137).

*Scalar types* A variable declared with integral or floating-point type is a *scalar*. A scalar type is usually a fundamental type, rather than a derived type.

### 5.4.5.1 Pointers

A *pointer* is a data item that contains an address. The address shows where to find another data item in memory. You designate a data item as a pointer by placing the *ptr\_operator* `*` before a *declarator*. For example:

```
int i;
int *p = &i;
*p = 3;           // Assigns 3 to i
```

This code declares `p` to be a pointer to an `int` and initializes it with the address of `i`.

#### Pointer assignment rules

The type pointed to can be any fundamental or derived type. A pointer must be initialized before it can be dereferenced. You can initialize a pointer by assigning an address to it, using the `new` operator, the `&` operator, or a function such as `malloc()` that returns an address. You can also assign one pointer to another. Here are some rules that apply to pointer assignments:

- If the left operand of an assignment is of a pointer type, the right operand must be of a pointer to an identical type or a constant expression that evaluates to `0` (zero).
- You cannot assign a pointer of type `const type1 *` to a pointer of type `type1 *`, but you can assign a pointer of `type1 *` to a pointer of type `const type1 *`.
- If the left operand of the pointer assignment is of pointer-to-member type, the right operand must be of pointer-to-member of an identical type, or a constant expression that evaluates to `0` (zero).
- A pointer to a member of `class1` can be assigned to a pointer to a member of `class2` of the same type, if `class2` is directly or indirectly derived from `class1` by an unambiguous public derivation.

See Chapter 6: *Expressions and Conversions* for more information about `new` and `&`, as well as how to access data using pointers.

### 5.4.5.2 References

You can think of a *reference* as being just like a pointer, except that when a reference variable is used in an expression, the pointed-to data item is fetched. References are restricted: you cannot have a reference to a reference, a reference to a bit field, arrays of references, or pointers to references.

You designate a data item as a reference by placing the *ptr\_operator* `&` before a *declarator*. For example:

```
int i;
int &r = i;
r = 3;           // Assigns 3 to i
```

*Initializing a reference* You initialize the value of a reference with any value whose address can be taken. A reference must *always* be initialized; you cannot declare a reference (except as a function parameter, structure member, or **extern** variable) without providing its initial value. Contrast this example with the pointer example in §5.4.5.1: *Pointers* on page 60.

*Call by reference* References are most often used for function parameter types — to implement “call-by-reference” — and return types. Stand-alone reference declarations are seldom used.

### 5.4.5.3 Arrays

An *array* is an object containing a group of items that occupy an area of storage. Each item in the array has the same size and data type.

*Accessing array elements* An array element can be of any fundamental or derived type, except a function. Array elements can be pointers to functions, however. You use subscripts to access individual elements of the array. A *subscript* is an integer expression enclosed in square brackets `[]` that specifies which element of an array is being accessed. The first element of an array is accessed with subscript `0` (zero). For example:

```
int iarray[10];
iarray[0] = 1;    // This assigns 1 to the first
                  // element of iarray.
iarray[9] = 10;   // This assigns 10 to the last
                  // element of iarray.
```

*Declaring an array* As in this example, you declare an array by using square brackets enclosing a compile-time-constant integer expression, after the identifier name. The example declares an array of 10 integers with the name **iarray**. Arrays are zero-based in C and C++.

*Arrays of arrays* C and C++ do not have true multi-dimensional arrays. Instead, you can declare an array of arrays. For example:

```
long myarray[10][20][40];
long i,j,k;
for (i = 0; i < 10; i++)
    for (j = 0; j < 20; j++)
        for (k = 0; k < 40; k++)
            myarray[i][j][k] = 0xffffffff;
```

In this example, `myarray` is declared as ten arrays of 20 arrays of 40 integers. The example code iterates, assigning the hexadecimal constant `0xffffffff` to each element of the array.

*Array expressions point to the first element* Except as an operand of `sizeof`, any expression of an array type is implicitly converted to a pointer to the first element of the array. This might seem like a strange rule, and it gives arrays their peculiar character in C as second-class citizens (you cannot pass them as parameters, assign them, and so forth). This rule is intimately tied in with the way the subscript operator works in C. `a[i]` means `*(a+i)`. `a` is converted to `&a[0]` by this rule, so that `a[i]` is finally equivalent to `*(&a[0]+i)`. This means: take the address of the first element of `a`, skip past `i` elements (`+i`), and dereference the result — the data item at the `i`th location of `a`.

#### 5.4.5.4 Structures

A *structure* is a collection of variables, possibly of different types, grouped together under a single name. Each variable in the structure is called a *structure member* or a *structure element*.

*struct and class* In C++, a `struct` is like a `class` that has all its members publicly accessible. See Chapter 9: *C++ Classes* for more information about classes. Usually, the elements of a structure are logically related in some way. For example, if you want to keep all information about each employee at your company collected under one data structure, you could define a structure that looks like this:

```
struct employee_rec {
    char    *name;
    int     social_security_number;
    float   hourly_wage;
    char    *date_of_hire;
    char    *job_title;
```

```

long    department_number;
double  vacation_hours;
double  sick_hours;
int     withholding_exemptions;
};

```

This series of declarations defines a structure type called `employee_rec`. All the different fields of an instance of the structure contain information about a particular employee.

*Defining structures* A *class\_definition* — which is a *type\_specifier* — allows you to define a structure. It is called a *class\_definition* because the forms of structure, union, and class definitions are essentially the same (classes are treated in depth in Chapter 9: *C++ Classes*).

A *class\_definition* is defined as follows:

```

[ _Packed | _Unpacked ] { class | struct | union }
    [ identifier ] [ base_list ]
    { { member_declaration | access_specifier : } ... }

```

#### Extension

`_Packed` and `_Unpacked` are High C/C++ extensions. You can use these keywords along with compiler toggles to control the alignment of structure members.

*Aligning structure members* If you use `_Packed`, and packed structures are supported for the target processor, structure members are not aligned. If you use `_Unpacked` (the default), structure members are aligned on “natural” boundaries. For example, a `long` member would be aligned on a long-word boundary on most 32-bit machines. See the **High C/C++ Programmer’s Guide** for your target system for more information.

---

**WARNING:** `_Packed` and `_Unpacked` are not implemented for all target platforms. See your `README` file for information.

---

*Declaring structure members* A *member\_declaration* is one of the following:

```

[ decl_specifiers ]
    [ member_declarator [ , member_declarator ] ... ];
function_definition [ ; ]

```

A *member\_declarator* is more than a simple *declarator*; it is one of the following:

```
declarator [= 0]
[identifier] : constant_expression
```

*Features unique to C++* *base\_list*, *access\_specifiers*, the presence of `= 0` in *member\_declarator*, and *function\_definition* in a *member\_declaration* are all features unique to C++. These features are treated in greater depth in Chapter 9: *C++ Classes* and Chapter 10: *Access Control in C++*.

To illustrate the similarities of the **class** to the **struct** and the **union** in C++, consider the following example:

```
struct type_t {
    private :
        char c;
    public:
        long a;
        void setchar(char *ch){c = *ch;}
};
```

*Access specifiers* This example declares a structure type. In C++, you can use *access specifiers* such as **private**, **protected**, and **public**, when declaring structure and union types, as well as when declaring class types. Notice also that *type\_t* contains a function definition. See Chapter 10: *Access Control in C++* for more information about access specifiers and member functions.

#### 5.4.5.5 Unions

A *union* is a collection of variables, including possibly structures and classes, that overlay one another in memory. You can effectively treat the same area of memory as though it is represented at different times in one of several different data types. You can declare unions inside structures, and you can have an array of unions. Unions can be structure members, also.

```
union fltpt{
    struct istruct{
        long word1;
        long word2;
    } iword;
```



```

struct fstruct{
    float fp1;
    float fp2;
} fword;
double dflt;
}

```

Access **union** members carefully

In this example, **ifword**, **fword**, and **dflt** all occupy the same memory. A union's size is determined by the size of its largest member. Each union element can be any valid C or C++ declaration. When you access a union member, you must retrieve the value using the type that was most recently stored.

---

**WARNING:** *Compilers do not enforce this restriction.* If you store a value into one union member and retrieve it from a union member of a different type, the results are implementation-dependent.

---

These are other aspects of a **union** that are important to remember:

- A **union** can be initialized, but only with a value corresponding to the type of its first member. See §5.7: *Initialization* on page 72 for more information.
- A **union** declaration cannot use the **\_Packed** or **\_Unpacked** modifiers.
- A **union** cannot have static data members.

#### C++ Only

Additionally, for C++:

- A **union** cannot have virtual functions, because there is no space for the virtual-function table.
- A **union** cannot have base classes, because there is no space for placing inherited base class members.
- A **union** cannot be used as a base class.
- A **union** cannot have a member whose type is a class with a constructor, destructor, or user-defined assignment operator. Imagine two members with constructors; which (overlapping) member gets constructed when a **union** object is created?

*Anonymous unions* An *anonymous union* is a union declaration of this form:

```
union { member_list };
```

An anonymous union declares an area of storage occupied by the union members in *member\_list*. The identifiers for anonymous union members must

be unique within the scope where the union is declared, because you can refer to them directly without the usual member-access syntax. If you declare an anonymous union at file scope, you must declare it **static**. In C++, you cannot declare members of an anonymous union as **protected** or **private**. In an anonymous union, you cannot declare function members.

```
main() {
    union {
        int i;
        char c;
    };
    // Now i and a are directly visible.
    i = 1;      // i is the int member of the union.
    printf("This machine is %s endian\n",
        c == 1 ? "little" : "big");
}
```

Anonymous unions are most often used within structure types, to obtain two or more structure members that overlap in storage:

```
struct s {
    enum {man, woman} gender;
    union {
        int with_child;
        int bearded;
    };
} person;
```

Here, if `person.gender == person::man`, `person.bearded` is relevant; otherwise, `person.with_child` is relevant.

#### 5.4.5.6 Bit Fields

A *bit field* is a union or structure member that is declared to occupy a certain number of bits. You can declare named and unnamed bit fields as union or structure members. An unnamed bit field is not accessible, but is useful as padding for alignment. A *member\_declarator* of the form:

```
[ identifier ] : constant_expression
```

specifies a bit-field declaration, where:

*constant\_expression* is a compile-time constant integer expression whose value is no greater than 32 (bits) on most computers.

*Use bit fields for Boolean members* A bit field is an efficient way to declare a Boolean structure member, whose value is effectively true or false.

An example of a bit-field declaration is:

```
struct {
    int bytel:8;
    int fourbits:4;
    int andtherest:20;
};
```

---

## 5.4.6 Type Qualifiers

A *type\_qualifier* is any one of the following:

```
const
volatile
_Far
_Near
_Alias
_Noalias
```

These are used to “modify” types. See §5.4.6.1: *Type Qualifier const* on page 67 and §5.4.6.2: *Type Qualifier volatile* on page 68 for more information about **const** and **volatile**, and §5.4.6.3: *Type Qualifiers \_Alias and \_Noalias* on page 69 for more about **\_Alias** and **\_Noalias**. See the **High C/C++ Programmer’s Guide** for more information about **\_Far** and **\_Near**.

### 5.4.6.1 Type Qualifier **const**

An object of a **const** type cannot be modified. Most attempts at doing so elicit compiler errors. You can use a cast to escape the restriction, but the result is undefined:

```
const int x = 3;           // x is forever 3.
const int *p = &x         // p points to a
                           //   const int (here, x).

func() {
    x = 4;                 // Compiler catches the error.
    *p = 4;                // Compiler catches the error.
    *(int *)p = 4;         // This might work or might not.
}
```

*Cast not recommended* The cast of `p` to a pointer-to-non-**const** is not recommended. The compiler might have placed `x` (to which `p` points) in read-only memory, and the assignment might produce a run-time hardware exception.

*Two meanings of **const*** Due to the C language's declarator syntax for pointer types, there are two possible places to write **const**, and they mean different things. The **const** immediately following the `*` means that the pointer itself is constant; the **const** preceding the `*` means that what is pointed to is constant.

```
const int *pci;           // Pointer-to-const-int
int *const cpi;          // const-pointer-to-int
const int *const cpci;    // const-pointer-to-const-int

func() {
    pci++;                // OK, pointer not constant
    (*pci)++;             // Error
    cpi++;                // Error, pointer constant
    cpci++;               // Error
    (*cpci)++;            // Error
}
```

*static const objects* In C++, a **static const** object of an integral type can be used wherever a constant expression is required. This can replace the frequent usage of **#define** in C to achieve the same thing. For example:

```
static const LIM = 10;
int a[LIM*2];
```

C ≠ C++
---------

In C++, in contrast to C, a **const** object declared in the global scope is assumed by default to be **static** unless you use **extern**.

#### 5.4.6.2 Type Qualifier **volatile**

An object of type **volatile** is assumed by the compiler to change via mechanisms unknown to the compiler. The compiler does not try to optimize away references to such objects. **volatile** is useful when external events such as timers modify memory while a program is running. For example:

```
func() {
    volatile int *p = get_clock_address();
    int time1 = *p;    // Has one value
    int time2 = *p;    // Might have another value
}
```

### 5.4.6.3 Type Qualifiers `_Alias` and `_Noalias`

When you declare a variable with the `_Noalias` type qualifier, you are telling the optimizer to assume that the variable cannot be accessed or modified through any indirect means, such as through a pointer dereference.

When you apply `_Noalias` to a pointer-based variable (for example, `_Noalias char *p`), the optimizer assumes that no other pointer references the same memory location at the same time, and that the pointer does not contain the address of any variable that is accessed in the same function.

---

**WARNING:** The compiler makes no attempt to confirm that these assumptions have not been violated. If the `_Noalias` qualifier is used incorrectly, the optimizer’s assumption is wrong and the compiler can generate incorrect code.

---

When you declare a variable with the `_Alias` type qualifier, you are telling the optimizer that the variable can be accessed by indirect means.

`_Alias` and `_Noalias` are mutually exclusive.

---

## 5.4.7 Defining Types with `typedef`

You can give a name to any type — fundamental or derived — with the `typedef` storage category. When you use `typedef`, you are declaring a synonym for that type. For example:

```
typedef char *CHARPTR;
```

This declaration makes `CHARPTR` a synonym for “pointer-to-`char`”. You can now use `CHARPTR` to declare objects of that type. For example:

```
CHARPTR iamapointertochar;
char *iamapointertochartoo;
```

You cannot use a type name that you define with another `type_specifier`, except for a `type_qualifier`:

```
typedef long int INTEGER32;
unsigned INTEGER32 i;    // This is not valid.
const    INTEGER32 j;    // This is OK.
```

## 5.5 Declarations and Precedence

In a declaration, you must pay attention to the precedence of the operators used in the *declarator*. The operators in a *declarator* are `*`, `&`, `[]`, and `()`. The `*` and `&` come from *ptr\_operator*, and `[]` and `()` from the alternatives for *declarator* (see §5.1: *Declarations* on page 48).

The operators used in a *declarator* have exactly the same precedence as operators in *expression*, with the exception that the `&` is treated the same as `*`. Thus there are really just two levels of precedence here: `[]` and `()` bind tighter than `*` and `&`.

To see the importance of precedence in declarations, consider the following example:

```
int **x[3]();    // Not valid! See below.
```

Here, `[]` and `()` bind tighter than `*`, so first we have `x[3]` — `x` is an array-of-three ...; then we have `x[3]()` — `x` is an array-of-three-functions. Then we have the two `**`s; `x` is an array-of-three-functions-of-pointers-to-pointers-to-`int`. This is not a valid declaration, because arrays cannot contain functions — though they can contain pointers to functions.

Another way to think of this is to pretend to “execute” the declaration as if it were an expression. In fact, the guiding principle for C declarations is that an identifier should be declared as it is used in an expression, so you can read the declaration as if you were reading an expression.

Treating `**x[3]()` as an expression, we first apply `[3]`, because it binds tighter than the `**`s, and so we treat `x` as an array, and access an element. `()` is applied next, so the element is a function. Then we apply `*` (dereference) twice and obtain an `int`. So, again, `x` is an array-of-functions-of-pointers-to-pointers-to-`int`.

*Use parentheses to override precedence* To make this example illustrate an array containing pointers to functions, use parentheses to override precedence, exactly the same way you use parentheses to override precedence in an expression:

```
int *(*x[3])();    // Valid
//   ^      ^      Parentheses added
```

Using the read-an-expression method, we take `x`, apply `[3]`, then dereference the result — the inner `*` — because the added `()`s have forced the `*` to be applied next. The result is called — `()` — and then we dereference again — the first `*`. So `x` is of type array-of-three-pointers-to-functions-returning-pointer-to-`int`. This *is* valid.

Just remember this in declarations: `*` and `&` are less binding than `[]` and `()`. So `*x[p]` is interpreted as `*(x[p])` instead of `(*x)[p]`.

## 5.6 Linkage Specifications

*Communicating  
with other  
languages*

A *linkage specification* is a mechanism C++ provides to enable your program to communicate with code that is not written in the C++ language. Functions declared with linkage specifications are **extern**. You can declare functions as **static** within a linkage specification, but the linkage specification will be ignored for those functions. By default, if no linkage specification is given for an external function declaration, C++ linkage is assumed. You can use a linkage specification only at file-scope level. See §5.2: *Blocks and Name Scoping* on page 49 for more information about scope levels.

*Specifying linkage*

The form for a linkage specification is one of the following:

```
extern language_name { [declaration] ... }
extern language_name declaration
```

where *language\_name* is a string literal that corresponds to the name of the programming language in which the external code is written.

*Linkage depends on  
language and  
implementation*

Linkage from C++ to other languages and from other languages to C++ is implementation- and language-dependent. A *language\_name* is a string that names another programming language. You cannot use a language name that High C/C++ does not recognize. High C/C++ recognizes the following language names:

```
"C++"
"C"
"Pascal"
"FORTRAN"
```

For example, to access C library function `strlen()`:

```
extern "C" int strlen(const char *);
```

See the **High C/C++ Programmer's Guide** for more information about mixed-language programming.

## 5.7 Initialization

An *initializer* following a declarator allows you to initialize data items in declarations. There are two forms of *initializer*. The first is from ANSI Standard C and the second from C++:

```
= init_value
(expression [ , expression ] ... )
```

where *init\_value* is one of the following:

```
expression
{ init_value [ , init_value ] ... [ , ] }
```

A single *expression* following = must be convertible to the type of the object being initialized. A brace-enclosed list of *init\_value* items is used to initialize a structure or array. For an array, each *init\_value* initializes a member of the array, starting from zero. For a structure, each *init\_value* initializes a data member of the structure, in the order the members are declared in the structure. If the structure is a **union**, only the first member is initialized.

Nested *init\_value* items can be used where a structure contains sub-structures or arrays as members; they can also be used when an array has elements that are sub-structures or arrays. Each expression in a brace-enclosed list must be convertible to the type of the array element or structure member being initialized.

*Initializing arrays* When you initialize an array, you can omit the size of the array in the array declarator. The size is then inferred from the number of elements in the brace-enclosed list. If you give the array size but do not list enough expressions to initialize all the array elements, the remaining elements are either initialized to 0 (zero) (if the element type has no constructor) or merely constructed (if the element type has a constructor). Listing too many expressions for the array size is an error.

*Initializing structures that have constructors* The brace-enclosed list cannot be used on structures that have constructors; the second (C++) form of initializer is provided for that. In some cases the



compiler provides a constructor for a structure type if you do not, depending on the members of the structure.

You can initialize values of **auto**, **register**, **static**, and **extern** objects. **static** variables without explicit initial values and without constructors are initialized to a default value of 0 (zero), converted to the appropriate type.

*Initializing static entities*

In C, initial values for **static** entities must be constants or constant expressions. In C++, this restriction is not present, but there is overhead when the expression is non-constant. The overhead consists of generated code that tests whether the **static** entity has been initialized. If it has not, initialization code is executed to initialize it.

*Initializing objects with constructors*

The second form of *initializer* — the parenthesized expression list — is used for objects of types with constructors. The listed expressions are taken as arguments to a suitable constructor for the object. If the object's type has no constructor, you can still use this form of *initializer*, but you can supply only one expression. The meaning is the same as if you had written *= expression*.

You cannot use the C++ form of *initializer* and supply *no* arguments to a constructor, because `classname X()` is instead interpreted as a function `X()` taking no arguments and returning a `classname`. If you just write `classname X`, the compiler will find the no-argument constructor anyway and call it.

*Scope of initializers*

An initializer for a static member of a class is in the scope of the class. Thus, any initializing expressions can use names declared in that class scope without using the scope resolution operator `::`.

---

**Note:** Overloading the `=` operator does not affect initialization. The symbol `=` used to introduce an initializer has nothing to do with assignment; both assignment and initialization just happen to use `=`.

---

Example 5.1 illustrates many kinds of initialization.

**Example 5.1 Various Kinds of Initialization**

---

```

int i = 3;
int &ref_i = i;
int *j = &i;
float fl[] = { 2.3, 6.2 };
// fl is [2], inferred from
// the length of the initialization list.
char st [3] = { 'e', 'f', 'g' };
struct astruct {
    int l, m; int a[2];
} thisstruct = { 1, 2, {3, 4} };
class boy {
public:
    int height, weight;
    boy(int i, int j){ height = i; weight = j;};
    boy(){ };
    static int count1;
    static int count2;
};
int boy::count2    = count1;
                // ^^^^^ Member of boy; no :: needed
union { int a; char b; } x = { 1 }; // Initializes a
void myfunc() {
    boy harry;                // Default constructor used
    boy tom = boy(65,163);    // Initialize class using
                                // constructor
    boy thom(65,163);         // Same as for tom, but
                                // see below
    boy tomas = {65,163};     // Error: class
                                // has constructors
}

```

---

*Constructors and initialization* For classes with constructors, there is a subtle difference between the two forms of initialization. In the first (C language) form:

```
boy tom = boy(65,163);
```

the value `boy(65,163)` is a single expression that can be converted to the type of `tom`. But in putting this value into `tom`, the class's copy constructor is used (a class always has a copy constructor; if you do not provide one, the

compiler does). So here (a) a constructor is called, putting `boy(65,163)` into a temporary; and (b) the temporary is copy-constructed into `tom`.

In the second form:

```
boy thom(65,163);
```

the constructor taking two `int` arguments is called directly, and it constructs its result into `thom`. This form of initialization for classes with constructors is, in general, more efficient.

Where the compiler can detect that a copy constructor does nothing more than a bitwise copy, it can often optimize away the copy and construct directly into the variable being initialized:

```
boy tom = boy(65,163);
```

becomes:

```
boy tom(65,163);
```

High C++ does this for `thom` in Example 5.1: *Various Kinds of Initialization* on page 74, which is why the comment in the example says the effect is the same. However, this is not true in general.

*Some C++ statements can be read as either declarations or expressions*

The C++ form of initialization together with C++'s function-style form of cast leads to C++ statements that can be interpreted as either declarations or expressions, and can make C++ programs hard to read. See §7.3.1: *Resolving Ambiguity* on page 101 for more on this subject.



# 6

## Expressions and Conversions

---

An *expression* is a sequence of data items and operators that specify a computation. There is no limit on the number of components in an expression. This chapter presents the elements of an expression and describes the order of their evaluation, discusses the use of operators in expressions, and discusses type casting and data-type conversions. Operator overloading is discussed in Chapter 12: *Name Overloading and Operator Functions in C++*.

---

### 6.1 Elements of an Expression

*Precedence* An expression is structured according to precedence rules. Table 3.3: *Precedence and Associativity of C++ Operators* on page 21 shows a list of C++ operators and their precedences. The evaluation order of independent subexpressions within an expression is undefined. If two sets of parentheses are at the same nesting level, the order of evaluation is undefined. For example:

```
int i, j[2];  
i = 3;  
j[i] = i++;
```

The expression `j[i] = i++;` could be evaluated either as `j[3] = 3;` or as `j[4] = 3;`.

*Use parentheses to override precedence* When you want to be sure of the evaluation order of an expression, you should break up the expression into its components. As another way to force a particular evaluation order, if you enclose a portion of an expression in parentheses, the subexpression within the most deeply nested pair of parentheses is evaluated first.

*Objects, lvalues, and rvalues* An *object* is a region of storage associated with a particular instantiation of a data item. An *lvalue* is an expression that refers to an object. Originally, lvalue meant “something that can appear on the left side of an assignment operator”. However, it is possible to have an lvalue that refers to a constant. An lvalue refers to a modifiable object if it is not a function, an array, or a declared `const`.

An expression such as `a*b` is not an lvalue. The expression does not refer to an object. Expressions of this kind are called *rvalues*. An expression can produce an lvalue, an rvalue, or no value. Even if an expression produces no value, it can cause a side effect.

---

**Note:** `a*b` does not refer to an object unless the `*` operator has been overloaded with such a meaning.

---

---

## 6.2 Conditional Expressions

A conditional expression uses the `? :` construct familiar to C programmers. This is the form of a conditional expression:

`logical_expression ? expression : expression`

Either of the *expression* operands can also be a conditional expression. The *logical\_expression* operand must have pointer or arithmetic type. *logical\_expression* is evaluated: if it evaluates to a non-zero value, the expression immediately following `?` is evaluated, and its result is the result of the conditional expression. If *logical\_expression* evaluates to 0 (zero), then the expression immediately following `:` is evaluated, and its result is the result of the conditional expression.

Side effects in *logical\_expression* must completely execute before either *expression* is evaluated. The results of the *expression* operands are converted to a common type if they happen to be of different types. See §6.5: *Conversions* on page 93.

---

## 6.3 Constant Expressions

A *constant expression* is an expression using any sequence of one or more valid C++ operands and operators, where each operand has an integral

constant value that is known when compiling your source program. You must use constant expressions for the following:

- array bounds
- **case** values (see §7.4: *Selection Statements* on page 102)
- bit-field lengths
- initial values for enumerators

If an operand in a constant expression does not have an integral type, it must be explicitly converted by a type cast to an integral type.

---

**Note:** You cannot use the comma operator or the assignment operator in a constant expression.

---

---

## 6.4 Operators

An expression often involves an operator or operators executing an action on one or more objects. Because you can overload operators (see Chapter 12: *Name Overloading and Operator Functions in C++*), when you encounter a particular operator such as `=` or `+`, the operator might have a meaning totally different from the usual meaning.

---

### 6.4.1 Assignment

The assignment operator is `=`. For example, the expression `e1=e2` can be read as “make `e1` have the same value(s) as `e2`”. Any expression appearing on the left side of `=` must be an lvalue that refers to a modifiable object.

An expression appearing on the left side of `=` *cannot* be any of the following:

- a function
- an unsubscripted array name
- a data entity that has been declared **const**

No member or any component of a member of a **struct**, **union**, or **class** appearing on the left side of `=` can be **const**.

*Rules for  
assignment  
operations*

For the assignment operation to be valid, one of the following conditions must be true:

- Both left and right operands of `=` are arithmetic types.
- If either operand is a **struct**, **union**, or **class** of type **T**, the other operand must also be of type **T**, unless an appropriate `=` operator is defined for **T** and for the type of the other operand.
- If either operand is a pointer, the other operand must be one of the following:
  - a pointer to the same type (except for **const** or **volatile** qualifiers)
  - a constant expression that evaluates to `0` (zero)
  - a pointer to **void**

You can use the assignment operator `=` in combination with other operators as a kind of shorthand notation. As an example, an expression of the form `e1 op= e2` means `e1 = e1 op e2`, where `op` is one of the following operators:

	>>	<<	+	-
%	&	/	*	^

*Assignment versus  
equality*

Do not confuse the assignment operator `=` with the operator that tests for equality, `==`. A common mistake for novice C and C++ programmers is to confuse the assignment operator for the equality operator by writing code such as the following infinite loop:

```
// This program contains an infinite loop.
main(){
    int i, j[2]
    for (i = 1; i = 3; i++){
        j[i] = 0;
    }
}
```

This program runs indefinitely. Presumably, `i==3` was intended.

*Assignment  
expressions*

Unlike some other programming languages, the C++ assignment operator is an expression, not a statement. You can use the assignment operator wherever an expression is valid. The type of an assignment expression is the type of its left side, and its value is the lvalue of the left side after the assignment is completed.



```
// This program fragment is valid.
int i,j,k;
f(&i);
// Assigns the value of j+2 to i, and test i
if ( i = j + 2)
    cout << "Hello";
k = i = j; // Assigns j into i, and then k
```

---

### 6.4.2 Comma

*Commas order expressions for evaluation* The comma operator “,” is a separator that orders expressions for evaluation from left to right. The values of all but the rightmost expression are discarded. All side effects in each expression are completed before the next operand is evaluated. The type and value of the expression list is the type and value of the rightmost expression.

*Commas in parameter lists are special* In some contexts (such as parameter lists) comma has special meaning, so you must enclose expressions in parentheses to obtain the normal comma semantics. For example:

```
fnc( (i=3, i+=2), j)
```

Here `fnc()` is passed two arguments: the comma expression `(i=3, i+=2)`, and `j`.

---

### 6.4.3 Addition and Subtraction

The operator `+` means addition in C++. The subtraction operator is `-`. The `+` and `-` operators are called *additive operators*. These operators perform the normal mathematical operations of addition and subtraction.

*Adding and subtracting arithmetic types and pointers* Operands of additive operators must be of pointer or arithmetic type, such as `int`, `char`, `float`, or `double`. You cannot add two pointers together, but you can subtract one pointer from another. If you add or subtract a pointer and a non-pointer, the result is a pointer, and the non-pointer’s value is multiplied by the size in bytes of the object that the pointer addresses.

For example:

```
double dar[10];  
double *p=dar;  
p = p + 2;
```

This example shows a pointer-to-**double**, **p**. The pointer is initialized with the address of the array **dar**, so that **p** points to the address of the first element of **dar**, **dar[0]**. The expression **p + 2** means “add the address of **dar[0]** and **2 \* sizeof(double)**”. After the assignment is completed, **p** points to the third element of the **dar** array. If you perform pointer arithmetic with an additive operator, and the resulting address is outside the bounds of the array, the result is undefined.

*Pointers as  
operands*

You can use two pointers as operands for the **-** operator, but the result is undefined unless they point to the same array object. The result is a type **int** value, not a pointer, that represents the number of elements separating the two pointer addresses. You cannot use a pointer-to-function or a pointer-to-**void** as an operand for an additive operator.

See §6.5: *Conversions* on page 93 for information about conversions and widening rules.

---

## 6.4.4 Multiplication, Division, and Modulus

A *multiplicative operator* is one of the following operators: **\***, **/**, or **%**. These are all binary operators. A *binary operator* has two operands, one on the left side, and one on the right side.

- Binary **\*** is the multiplication operator.
- Binary **/** is the division operator.
- **%** is the remainder operator.

Both **\*** and **/** must have arithmetic-type operands. Operands for **%** must have an integral type. Operands and results are converted to other types as necessary. See §6.5: *Conversions* on page 93.

If the right operand of either **/** or **%** is **0** (zero), the result of the operation is undefined. For **%** and **/**, the first operand is divided by the second. If either operand of **%** is negative, the sign of the result is hardware dependent.

## 6.4.5 Relational Operators

The relational operators perform comparisons. Relational operators are binary operators. Table 6.1 describes the relational operators available in C++.

**Table 6.1 C++ Relational Operators**

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Relational operators associate from left to right, but you must be cautious when using several of them without grouping parentheses. For example:

```
i < j < k
```

means:

```
(i < j) < k
```

It does *not* mean that `j` is less than `k` but greater than `i`.

*Precedence* Operators `==` and `!=` have lower precedence than the other relational operators. They are sometimes referred to as *equality operators* rather than relational operators.

*Comparing operands of different types* Operands of relational operators must have pointer or arithmetic types. You cannot use relational operators to compare entire aggregates, such as structures, classes, or unions. You must compare individual components of aggregates. If you compare operands of different types, the operands are converted to the same type before comparison. See §6.5: *Conversions* on page 93.

*Comparing pointers* Comparison of two pointers `p1` and `p2` is defined only in the following cases:

- `p1` and `p2` have the same type
- either `p1` or `p2` has type `void *`

If two pointer types do not satisfy these restrictions, implicit pointer conversions can be used to coerce one type into the other so that the types are equal and the results can then be compared. For example, you can compare a pointer to a base type and a pointer to a derived type because an implicit conversion converts the second pointer to the type of the first. See §6.5.4: *Pointer Conversions* on page 95.

You can compare a pointer with a constant expression that evaluates to `0` (zero).

*Comparing  
pointers-to-data-  
members*

You can compare pointers to data members of a **class**, **struct**, or **union**. For a **class** or **struct**, a pointer to a member `m1` is less than a pointer to a member `m2` in the same object if the declaration of `m1` textually precedes the declaration of `m2`, unless the two members are separated by an access qualifier such as **public**, **private**, or **protected**. Do not compare two pointers to non-**static** members that are separated by an access qualifier, because the result of the comparison is undefined.

---

## 6.4.6 Shift Operators

C++ has two shift operators, `<<` (left shift) and `>>` (right shift). Shift operators are binary operators, and are interpreted as “shift the left operand to the right or left (depending on the operator) as many places as the integer value of the right operand”. Both operands must be of integral type. For left shift, the vacated bit positions are filled with zeros. For right shift, if the left operand is declared **unsigned** or has a positive value, vacated bit positions are zero filled. Right shift of a signed negative value is filled with ones.

*Divide or  
multiply by 2*

Right shift has the effect of dividing by 2 for each bit position shifted. Left shift has the effect of multiplying by 2 for each bit position shifted. The result of the shift operation is undefined if the shift count (right operand) is negative. The result is also undefined if the value of the right operand is greater than or equal to the number of bits in the left operand.

---

### 6.4.7 Logical AND and OR

*Chaining expressions* You can chain together logical or relational expressions to form complicated logical expressions using the binary `||` (logical OR) and `&&` (logical AND) operators.

*Result of a logical operation is 1 or 0* The result of an expression using these logical operators is `1` (one) or `0` (zero).

- For a logical OR, the result is `1` (one) if either operand is non-zero.
- For a logical AND, the result is `1` (one) if both operands are non-zero.

These two operators associate left to right, so that in an expression like `b || c || d`, the subexpression `b || c` is evaluated first, followed by `(b || c) || d`.

C++ endorses “short-circuit” evaluation of logical AND and OR: if the result of the operation can be determined after evaluating only one operand, the other operand is not evaluated.

---

### 6.4.8 Bitwise AND, OR, and Exclusive OR

Bitwise operators differ from logical operators in that each bit of both operands is evaluated, rather than simply considering whether each operand is `0` (zero) or a non-zero value. With a logical operator, if any bit of an operand is nonzero, the entire operand is considered a non-zero value. For a bitwise operator, each bit in an operand is individually considered.

*Operands must be integral* C++ offers bitwise AND (`&`), bitwise OR (`|`), and bitwise exclusive-OR (`^`) operators. Each of these operators is a binary operator. Operands of bitwise operators must be integral. Type conversions are performed as necessary among integral types. See §6.5: *Conversions* on page 93.

Bits in each operand are compared sequentially, evaluating each bit position in one operand with the corresponding bit position in the other operand. The following results occur for each of the bitwise operators:

- A bitwise AND results in a bit position set to **1** (one) if the bit position in both operands is a **1** (one). Otherwise, the bit position is set to **0** (zero) in the result.
- A bitwise OR results in a bit position set to **1** (one) if the same bit position in either operand is a **1** (one). Otherwise, the bit position is **0** (zero) in the result.
- A bitwise exclusive OR sets a **1** (one) in the result for each bit position where the operands differ. If the bit position has the same value in both operands, the bit position is set to **0** (zero) in the result.

---

## 6.4.9 Pointer-to-Member Operators

C++ provides two pointer-to-member operators, `.*` and `->.`. Both operators are binary operators. The operator `.*` allows you to access a member in a particular object through a pointer.

*Left operand of `.*`  
must be a class;  
right operand must  
point to a member*

The left operand of `.*` must be a class containing the member to which the right operand points, or it must be a class derived unambiguously from an accessible base class that contains the member to which the right operand points. A base class is *accessible* if the context allows access to the base class's public members as inherited by the derived class.

The right operand of `.*` must be a pointer to a member in a class.

*Left operand of `->.*`  
must point to a class  
object; right  
operand must point  
to a member*

The operator `->.*` is similar to `.*`, but in this case, the left operand must be a pointer to a class object, rather than an actual class object or class type name. Both operands of `->.*` are pointers. The left operand is a pointer to a class object. The right operand is a pointer to a member of the class object addressed by the left operand, or it is a pointer to a class object for which the dereferenced left operand is an unambiguously derived and accessible base class.

If the member object to which the right operand points is a function, you can use the result of the pointer-to-member expression only as an operand to the function-call operator `( )`. If the operand of the pointer-to-member expression is an lvalue, the result of the pointer-to-member expression is an lvalue.

See §9.10: *Pointers to Members* on page 148 for an example of how to use pointer-to-member operators.

## 6.4.10 Unary Operators

These are the unary operators in C++:

`&`   `*`   `+`   `-`   `~`   `!`   `++`   `--`  
`sizeof`   `delete`   `new`

A unary operator has one operand.

### 6.4.10.1 Address Operator

*Return the memory address*

The address operator `&`, when placed before a data item, means “return the memory address of this item”. You cannot take the address of a data item declared with the `register` qualifier. You can take the address of a function. If the type of a *data\_item* is *t*, the type of `&data_item` is “pointer to *t*”.

*Address of an overloaded function*

You can take the address of an overloaded function, but only in an initialization or assignment where the left side of the assignment uniquely determines which version of the overloaded function is referenced. See §12.1.3: *Function-Match Overload Resolution* on page 196.

### 6.4.10.2 Dereference Operator

The dereference operator `*` means “take the expression to the right, treat it as an address, and reference the object at that address”. If *p* is a pointer to an object of type *o*, then `*p` yields the object at the address contained in *p*. The operand of `*` must be a pointer, and the result is an lvalue of type *o*.

### 6.4.10.3 Unary Plus

**C++ Only**

The unary plus operator `+` must have an operand that is of arithmetic or (for C++ only) pointer type.

The result of unary plus is the value of its operand — it effectively does nothing. For a pointer type, the type of the result is the type of the operand. For an arithmetic type, the type is that obtained after integral promotion (see §6.5.2: *Integer Conversions* on page 94).

#### 6.4.10.4 Unary Minus

The operand of a unary minus operator `-` must be of arithmetic type. Unary minus negates its operand after applying integral promotion. If the operand is unsigned, the result is found by subtracting the operand from  $2^n$ , where  $n$  is the number of bits in the operand. The type of the result is the type of the operand after integral promotion.

```
int i = 3;
int j = -i;           // j is -3.
unsigned k = 3;
unsigned l = -k;      // l is awfully big.
```

#### 6.4.10.5 Bitwise NOT

The `~` operator performs a bitwise NOT operation, taking the one's complement of its operand after integral promotion. Each bit in the result is set if the corresponding bit in the operand is not set. The type of the result is the type of the operand after integral promotion. For example:

```
short i = 0x35; // This is 0000_0000_0011_0101 binary.
short j = ~i;   // Result: 1111_1111_1100_1010 binary.
```

#### Extension

Notice the underscores in the binary numbers, to make them more readable. This is a High C/C++ extension. In High C/C++ you can actually input binary numbers with underscores, such as `0x2x0000_0000_0011_0101`. See §3.8.4: *Hexadecimal Constants* on page 28.

#### 6.4.10.6 Logical NOT

The logical NOT operator `!` accepts an integral operand, and the result is an integral value. If the operand has the value `0` (zero), the result is `1` (one). If the operand is a non-zero value, the result is `0` (zero).

#### 6.4.10.7 Pre-Increment and Pre-Decrement

#### C ≠ C++

An operand preceded by `++` or `--` is a *pre-increment* or *pre-decrement* expression. The operand must be an lvalue of a scalar type. The result is a non-lvalue of the same type. For `++`, `1` (one) is added to the lvalue; for `--`, `1` (one) is subtracted. For pointer types, `sizeof(pointed_to_type)` is added or subtracted.



---

**Note:** The ARM says pre-increment and pre-decrement are lvalues, but C++ may be made to conform to ANSI C in this area.

---

The increment or decrement occurs before the result is used as an operand in any surrounding expression. For example:

```
int i = 3;
char a[] = {'a','b','c','d'};
// i is decremented to 2 before subscripting.
if (a[--i] == 'c') printf("OK");
```

#### 6.4.10.8 Post-Increment and Post-Decrement

An operand followed by `++` or `--` is a *post-increment* or *post-decrement* expression. The operand must be an lvalue of a scalar type. The result is a non-lvalue of the same type. For `++`, 1 (one) is added to the lvalue, and for `--`, 1 (one) is subtracted. For pointer types, `sizeof(pointed_to_type)` is added or subtracted.

*Side effects and sequence points*

The value of the increment or decrement expression is that *before* the increment or decrement takes place. The increment or decrement is a side effect that occurs before the next “sequence point”. The *sequence points* in C++ are the completion of the evaluation of the following:

- the arguments to a function call
- the first argument of `&&`, `||`, `?`, or `,`
- an expression not used as a operand of another expression or as a function argument

It is common practice to write `i++`; instead of `i = i+1`;. Similarly for `--`.

Because the side effect can occur at any time up to the next sequence point, using postfix `--` or `++` as an operand of an expression can sometimes yield expressions with different results. For example:

```
int i = 3;
// i is decremented to 2 sometime before the ?.
printf(i-- == i
      ? "after second fetch"
      : "before second fetch"
      );
```

Here `i` is decremented before the `?`, but the decrement can occur before or after the second fetch of `i` (`== i`) occurs.

#### 6.4.10.9 The `sizeof` Operator

The `sizeof` operator gives the size, in bytes, of its operand. The operand can be an *abstract\_declarator* denoting a type, or it can be an expression. If the former, it must be enclosed in parentheses; if the latter, the expression is *not* evaluated.

```
printf("The size of an integer is %d\n",sizeof(int));
```

This example prints the size of an `int` on the machine where it is executed. See the **High C/C++ Programmer's Guide** for information about the possible operands for `sizeof` and the result for each kind of operand.

#### 6.4.10.10 The `delete` Operator

*Destroys objects created by `new`* The `delete` operator destroys an object that was created dynamically via `new`. The type of the result of a `delete` expression is `void`. The operand of `delete` must be a pointer that was created by `new`. The effect of using any other operand with `delete` is undefined. You can always delete a pointer whose value is `0` (zero), however, with no error.

You cannot access an object after it has been deleted. The effect of such an operation is undefined. A pointer to a `const` object can never be the operand of `delete`. If the pointer operand of `delete` points to an object that has a destructor, `delete` invokes the destructor.

This is the syntax for `delete`:

```
[ :: ] delete [ [ ] ] cast_expression
```

where *cast\_expression* is one of the following:

```
unary_expression  
( type_name ) unary_expression
```

*Deleting arrays* The form `delete[] cast_expression` is for deleting arrays. You must use this notation when deleting an array, or the result is undefined. You must *not* use this notation when deleting non-array objects, because this also gives undefined results.

*Class-specific delete operator* If the deleted expression points to a class object, the class-specific **delete** operator, if present, is used. See §11.4: *Operators new and delete* on page 184 for how to declare a class-specific **delete** operator. The leading **::**, if used, means that the class-specific **delete** operator is *not* used; instead, the global operator **delete** is.

#### 6.4.10.11 The **new** Operator

*Creates objects* The **new** operator creates an object of type *type\_name* and returns a pointer to the object.

The syntax for **new** is one of the following:

```
[ :: ] new [ other_args ] type_name [ ( initializer ) ]
[ :: ] new [ other_args ] ( type_name ) [ ( initializer ) ]
```

where the components have the following meanings:

*other\_args* is an expression or list of expressions separated by commas, supplying additional arguments to **new**. You can use this only when you have declared an **operator new()** function with appropriate arguments. For example, **new** results in a call to **operator new(sizeof(T))**, whereas **new(a, b+3)** results in a call to **operator new(sizeof(T), a, b+3)**.

*type\_name* is any fundamental or derived type name.

*initializer* is an expression or list of expressions appropriate to initialize the object you are allocating. See §5.7: *Initialization* on page 72.

*Allocating arrays* When you allocate an object of array type, all dimensions except the first must be compile-time constant expressions. **new** allocates storage for an object of *type\_name* equal to the size in bytes for **sizeof(type\_name)**. If *type\_name* refers to a class type or an enumerated type, the type must have been previously declared. You cannot declare a class or enumerated type in a **new** expression. *type\_name* cannot contain **const** or **volatile** qualifiers in its declaration.

```

int i = 3;
// Allocate an int [3][6][9] array:
int ar = new int [i][6][9];
char *str = "hello";
// A common use of new:
char *p = new char [strlen(str)+1];

```

If **new** is unable to allocate the requested storage, it returns 0 (zero).

*Initializing objects  
with new*

If you do not supply *initializer*, the allocated object has an undefined value, unless there is a constructor for it. Initialization is performed only when **new** returns a non-zero value. You cannot initialize objects of array type. You can use **new** to create an object that is an array of a class type only if the class type defines a default constructor. If there is a default constructor for the class type, the constructor is called for each element of the array.

**new** invokes constructors for types that have declared them. The order of evaluation for both the **new** operator and arguments to constructors is undefined.

If the type allocated is a class, the class-specific **new** operator, if present, is used. See §11.4: *Operators new and delete* on page 184 for how to declare a class-specific **new** operator. If the leading **::** is present, the class-specific **new** operator is *not* used; the global operator **new** is used, instead.

```

int *i;
class init{
    float f;
    char c;
};
init *myclass;
// Allocate an integer and initialize it to 7:
i = new int(7);
// Allocate a class object and initialize it:
myclass = new (init);

```

## 6.5 Conversions

An expression might have to be converted to another type when any of the following occurs:

- It has a data type different from that of the other operand of a binary operator.
- The expression is passed as an actual parameter to a function.
- It does not have the data type required for a particular operator.

*Implicit conversions* An *implicit conversion* manipulates an object and performs any action necessary to derive an object of a needed data type from an object of a different data type, without any type casting in the program source code to force the conversion. These conversions take place “automatically”.

For example, when a floating-point value is passed to a function, the value is automatically converted to type `double` if the type of the corresponding parameter is not known (such as in C, when there is no function prototype, or in C++, when `...` is used in the prototype).

### 6.5.1 Floating-Point Conversions

If you use `float` in a context that requires `double` or `long double`, the `float` value is converted to either `double` or `long double` as needed. Converting `float` to `double` or `double` to `long double` causes no loss of precision, and no change in the numerical value. `long double` is generally more precise than `double`, which is also generally more precise than `float`. When you cause a conversion from a more precise type to a less precise type, the result is either the next higher or the next lower representable value. If the value is out of range for the less precise type, such as having an exponent that is too large, the result of the conversion is undefined.

*Converting to an integral type* If you cause the conversion of a floating-point type to an integral type, the fractional portion of the floating-point value is truncated. If the floating-point value is not representable as an integral type, the result of the conversion is undefined.

*Converting from an integral type* If you use an integer in a context that requires a floating-point type, the integer is converted to the appropriate type with as much accuracy as your hard-

ware permits. Some integral values cannot be exactly represented as a **float**; most hardware can represent integral values as **double** values. If the integral value cannot be exactly represented as a floating-point value, loss of precision can occur. Rounding to the next higher or lower representable value depends on your hardware. See the **High C/C++ Programmer's Guide** for more information.

## 6.5.2 Integer Conversions

*Integral promotion and widening* When you use a value of type **char** or **short int**, an enumerator, or an **int** bit field, the value might have to be converted to **int** or **unsigned int**. This is called *integral promotion*. When a value is promoted, the size of the object is usually changed. Usually it is *widened*, because **int** or **unsigned int** objects are typically larger than the objects that are being promoted. The widened object must retain the same numerical value that it had before promotion.

*Converting to an unsigned type* If you convert an integer value to an unsigned type, the resulting value is the smallest unsigned integer that is congruent to the signed integer modulo  $2^n$ , where  $n$  is the number of bits used to represent the unsigned data type.

## 6.5.3 Arithmetic Conversions

Some binary C++ operators cause conversions of their two operands. These conversions, known as *arithmetic conversions*, are summarized in Table 6.2.

**Table 6.2 Arithmetic Conversions**

Operand 1 Type	Operand 2 Type	Result Type
<b>long double</b>	Any integral or floating-point type	<b>long double</b>
<b>double</b>	Any numerical type except <b>long double</b>	<b>double</b>
<b>float</b>	Any numerical type except <b>long double</b> or <b>double</b>	<b>float</b>
<b>char, short, enum</b>	<b>int, unsigned int, signed int</b>	<b>int, unsigned int, signed int</b>
<b>unsigned long</b>	Any integral type	<b>unsigned long</b>

**Table 6.2 Arithmetic Conversions (Continued)**

Operand 1 Type	Operand 2 Type	Result Type
<code>long int</code>	<code>unsigned int</code>	<code>unsigned long int</code>
<code>long</code>	Any integral type other than <code>unsigned long</code>	<code>long</code>
<code>unsigned</code>	Any integral type other than long or unsigned long	<code>unsigned</code>

### 6.5.4 Pointer Conversions

A pointer to an object of a given type can be converted to a pointer to an object of a different type.

A constant expression whose value is `0` (zero) can be converted to the null pointer. The null pointer must not be the same as a pointer to any object.

A pointer to an object that is not declared `const` or `volatile` can be converted to a pointer to `void`.

A pointer to an object of a derived class can be converted to a pointer to a base class, provided you can unambiguously determine which base class (there might be multiple occurrences with multiple inheritance), and provided the base class is accessible in the context of the conversion (see §6.4.9: *Pointer-to-Member Operators* on page 86).

The result of the conversion is a pointer to the base-class sub-object of the derived class; however, the null pointer (zero) retains its value through this conversion. This introduces some expense: if the conversion would require adding an offset to the derived-class pointer, the compiler generates code to test for `0` (zero) before performing the addition.

A pointer to a member of a base class can be converted to a pointer to a member of a derived class if the implicit conversion from pointer-to-derived to pointer-to-base would be allowed. This conversion is backward relative to the derived-to-base object pointer conversion, but there is a good reason. For example:

```
struct B { int xi };
struct D : B { int yi };
D *dp; B *bp;
```

```

int D::*dip = &D::y;
int B::*bip = &B::x;
bp = dp;          // OK
dp = bp;          // Error
dip = bip;        // OK
bip = dip;        // Error

```

`bp = dp` is reasonable because whenever you have a pointer-to-derived, the base object must exist somewhere inside the derived object, and so you can convert the pointer to a pointer-to-base. The reverse conversion `dp = bp` is not allowed implicitly because, given a pointer-to-base, there is no guarantee that the base object is a part of the derived object. The reverse conversion can be obtained by using a cast (see §6.5.5: *Type Casting* on page 96).

`dip = bip` is reasonable because `bip` is essentially an offset into a `B` structure that identifies the member `x`; in ANSI Standard C parlance, it is essentially `offsetof(B,x)`. Because `D` contains all members that `B` contains (by inheritance), it is possible to convert that offset value into an offset from `D` to `x`, that is, `offsetof(D,x)`. The reverse conversion `bip = dip` is not allowed implicitly because there is no guarantee that the member to which `dip` points exists in `B`; `dip` may be pointing to member `y`, for example, which does not exist in `B`.

See §9.10: *Pointers to Members* on page 148 for more information about pointers-to-members. The reverse conversion can be obtained by using a cast (see §6.5.5: *Type Casting* on page 96).

C ≠ C++
---------

In C programs, a pointer to `void` can be converted to a pointer to any other data type. This is not true for C++. A pointer to `void` must be explicitly cast to the other type. See section §6.5.5: *Type Casting*.

---

## 6.5.5 Type Casting

*Type casting* is a method of explicitly specifying a conversion from one data type to another. There are two forms of a cast:

```

( type_name ) expression
type_name ( expression_list )

```

where `type_name` is any fundamental or derived type.



The first form is from C and allows a single *expression* to be converted to any type. The second form is from C++. The *type\_name* is restricted to a single identifier, but more than one expression can be supplied; this is useful for calling class constructors directly.

When an explicit type conversion is also allowed implicitly, the rules and results are the same. An explicit conversion can perform conversions that cannot be done implicitly, however.

*Explicit conversion  
by casting*

The following explicit conversions by casting are permitted:

- Any conversion that is valid as an implicit conversion is valid as an explicit conversion, and has the same meaning.
- Any pointer can be explicitly converted to an integral type, provided the integral type is big enough to hold it.
- An integral value can be explicitly converted to a pointer. If you convert a pointer to an integer, and then back to a pointer, the result is the same as the original pointer. This is true for most architectures except for those where the size of an integer is smaller than the size of a pointer (for example, the 16-bit 286).
- A pointer can be converted to a pointer to a different data type. However, addressing exceptions can occur if the new data type has alignment requirements different from those of the previous data type. You can always cast a pointer to an object to a pointer to a smaller object, and back again, without changing the value of the pointer. You can explicitly convert a pointer to a function to be a pointer to an object if the object pointer type is large enough to hold the function pointer. Otherwise, the same rules apply for explicit and for implicit pointer conversions.

**Note:** On segmented architectures such as the 386, the size of pointers may be different from the size of long integers, and even two pointers may be of different sizes. On other architectures, pointers are always the same size.

- A pointer to a **const** type can be cast to a pointer that is not of a **const** type. If you perform this conversion, the pointer still addresses the original **const** object. If you try to modify the **const** object through the pointer resulting from the cast, the result is implementation dependent.

- A pointer to an object type can be cast into a pointer to a function if the pointer to the function type is big enough to hold the pointer to an object type.
- You can explicitly convert a pointer to a function of one type to a pointer to a function of a different type.
- You can convert an object to a class type only if you have declared an appropriate constructor or conversion function. The argument to the cast is taken as an argument to the constructor or conversion function.
- You can cast a pointer to a member to a different pointer-to-member type if both types are pointers to members of the same class.
- A pointer to a member of a derived class can be converted to a pointer to a member of a base class if the conversion in the other direction is allowed implicitly. The effect is the semantic inverse of the other conversion, and assumes that the first pointer points to a member of the base class; otherwise the semantics are undefined. See §6.5.4: *Pointer Conversions* on page 95 for the reasoning behind this assumption.
- You can cast a pointer to a base class to a pointer to a derived class, assuming the conversion in the other direction is implicitly allowed, and assuming that the base class is not a virtual base of the derived class. The effect is the semantic inverse of the other conversion, and assumes that the first pointer points to an object that is a sub-object of the derived class. See §6.5.4: *Pointer Conversions* for the reasoning behind this assumption.

*Converting a  
pointer-to-class*

If you cast from one pointer-to-class to another pointer-to-class, and the classes are not yet defined, the cast does not change the pointer value. However, the same conversion occurring later might in fact add a delta to the pointer if one class is a base of the other, to reflect the location of the base object as a sub-object of the derived object. This difference in code generated for the two identical casts can produce anomalies. You should avoid such casting before classes are defined.

# 7

## Statements

This chapter discusses statements, which control the way your program executes.

**C ≠ C++**

C++ has the same statements as ANSI Standard C, with one difference. In ANSI Standard C, you must place all declarations in a block before the first statement in the block. C++ allows you to place declarations anywhere they are needed, promoting the idea of declaring data items textually close to their first use.

**Extension**

High C allows you to intermix statements and declarations as an extension to ANSI Standard C, even when you are not using C++ features. Table 7.1 shows the statements you can use in your C++ program.

**Table 7.1 C++ Statements**

Statement	Description
<code>if</code>	Allows conditional execution of one or more statements and expressions
<code>switch</code>	Allows selection from one or more alternative execution paths, based on the value of an expression
<code>do...while</code>	Repetitively executes one or more statements until the condition associated with <code>while</code> is 0 (zero)
<code>for</code>	A loop construct that executes one or more statements or expressions while a condition holds true, often a counted number of times
<code>while</code>	Evaluates a logical expression and executes one or more statements and expressions while the value of the logical expression is non-zero
<code>break</code>	Causes execution to jump to the next statement or expression following a loop or <code>switch</code> statement
<code>continue</code>	Causes execution to jump to the end of the current loop, where the iteration condition, if any, is evaluated

Table 7.1 C++ Statements (Continued)

C ≠ C++

Statement	Description
<code>return</code>	Stops execution in the current function and causes execution to resume in the calling function, if any, possibly returning a value
<code>goto</code>	Causes execution to jump to a specific labeled point, bypassing any statements or expressions between <code>goto</code> and the label
<code>declaration</code>	Is a statement; see Chapter 5: <i>Declarations and Definitions</i> .
<code>expression</code>	Evaluates an expression for its side effects

## 7.1 Labeled Statements

A *labeled statement* is any valid C or C++ statement preceded by a statement label. A statement label has this form:

`identifier:`

Labels are targets of `goto` statements

Statement labels are used only to designate a statement as a target of a particular `goto` statement. You can use a label in a `goto` statement before you define the label. The `identifier` that precedes the colon in a label must be unique among labels in a function. Labels exist in their own scope within a function, and do not interfere with any other identifiers. `switch` statements have special `case` and `default` labels that you can use only within a `switch` statement. See §7.4: *Selection Statements* on page 102.

## 7.2 Expressions as Statements

An *expression statement* is any valid C++ expression followed by a semicolon. If you omit the expression and use just a semicolon, you have created a null statement.

Most statements in a C or C++ program, such as assignments and function calls, are expression statements. When an expression statement executes, all side effects from the evaluation of the expression must be completed before the next statement begins execution.

## 7.3 Compound Statements or Blocks

A *compound statement* or *block* is any list of one or more statements enclosed in curly braces `{ }`. When you enclose a group of statements in curly braces, you cause that group of statements to be treated as if it were a single statement. For example:

```
if ( i == 3 ) {
    j = 4;
    k = 5;
}
```

If `i` is equal to `3`, both assignments enclosed in the curly braces are executed. If `i` is not equal to `3`, neither assignment is executed.

### 7.3.1 Resolving Ambiguity

#### C++ Only

The definition of C++ contains some cases in which it is difficult to distinguish between an expression statement and a declaration, and even some cases where text could be either. These cases occur when a fundamental or derived type name is followed by a left parenthesis. The problem in C++, which does not occur in C, is that a type name followed by a left parenthesis can be a function-style type cast. For example:

```
typedef struct { int age; } *recptr;
recptr(X)->age = 0;           // This is an expression.
recptr(Y) = 0;               // This is a declaration.
int(i);                      // This is the declaration of i.
int(j) + 0;                  // This is an expression.
char(*string)(int);          // This is a declaration.
// f is a pointer-to-function taking a double.
double(*f)(double(x));       // This is a declaration.
// d is a pointer-to-double initialized with
// double(3).
double(*d)(double(3));       // This is an invalid
                             // declaration.
```

If a text can be taken as declaration, it is so taken; otherwise, it is taken as an expression statement.

---

**Note:** The statement `double (*d)(double(3))` is cited in the **Annotated C++ Reference Manual** as a declaration. However, that is not correct, as confirmed by Bjarne Stroustrup.

---

*How to tell an expression statement from a declaration* You can resolve the ambiguity by examining whatever follows the right parenthesis. If the token following the right parenthesis is an operator (such as `->`), a function-call operator with a value as the next token, or an initializer, the text is an expression statement. If the tokens following the left parenthesis are valid for a declarator, the text is a declaration. See Chapter 5: *Declarations and Definitions* for more information about declarators.

---

## 7.4 Selection Statements

A *selection statement* evaluates an expression and selects the next statement to be executed depending upon the value of the expression. `switch` and `if` are selection statements.

---

### 7.4.1 `switch` Statement

A `switch` statement has this form:

```
switch ( expression ) sw_statement
```

where the components have the following meanings:

*expression* is any valid C++ integral expression.

*sw\_statement* is either a compound statement or a *case\_label\_statement*. Usually it is a compound statement that contains one or more *case\_label\_statements*.

*case\_label\_statement* is one of the following:

```
[ case constant_expr [ .. constant_expr ] : statement ]
[ default : statement ]
```

When your program executes a `switch` statement, *expression* is evaluated. No two *constant\_expr* expressions in `case` labels can have the same value in a single `switch` statement. Execution transfers to the `case` label whose *constant\_expr* has the same value as the result of *expression*.

Declarations in `sw_statement` cannot be a declaration. However, if `sw_statement` were a `switch` statements compound statement, it could contain declarations. If such a compound statement contains declarations with initial values, you cannot position those initializations so that transfer of control to a `case` label skips over the initialization. For example:

```
switch (j) {
    int k = 3;      // Error
    case 3:
        int m = 4;
    default:
        if ( m == 4 ) // Error
}
```

Thus, declarations with initializers must appear in an inner block within the `switch` statement.

Ranges in case labels

#### Extension

As an extension to C and C++, High C/C++ permits you to specify a range of values for a case label. The first `constant_expr` before the ellipsis (`..`) is the *lower bound* of the `case` label. The second `constant_expr`, which follows the ellipsis, is the *upper bound* of the `case` label. When you specify a range for a case label, the case label matches `expression` whenever the value of `expression` is greater than or equal to the lower bound, and less than or equal to the upper bound of the range. You must not specify a lower bound whose value is greater than the upper bound of a range. You must not specify another case label in the same `switch` statement whose value is within the range.

```
void myfunc(int *p){
    *p = 3;
}
main(){
    int i;
    myfunc(&i);
    switch(i){
        case 7:          // This is valid.
            cout << "Reached 7.\n";
            break;
        case 2..4:       // This is valid.
            cout << "Found range of 2..4.\n";
        case 6..5:       // This is an error: the lower
                        // bound exceeds the upper bound.
```

```

        case 3:           // This is an error, because
                           // 3 is in the range 2..4.
        default:
            break;
    }
}

```

**default label** Each **switch** statement can have at most one **case** label named **default**. If no other **case** label in the **switch** statement matches the value of *expression*, execution transfers to the first statement following the **default** label. A **switch** statement is not required to have a **default** label.

If a **switch** statement does not have a **default** label, and the value of *expression* does not match any **case** label in the switch statement, no statement in the **switch** statement executes, and execution continues with the first statement following the **switch** statement.

**Flow of control** Once the statement or statements following the selected **case** label finish executing, execution continues with the statement following the next **case** label, unless you have coded a statement that alters the flow of control statement. This is a common error for novice C programmers who are used to programming in Pascal. In the preceding example, **case 7:** ends with a **break** statement. This alters the flow of control, transferring it to just past the end of the **switch** statement. But after **case 2..4:** is over, flow of control would pass into **case 6..5:** (if the example represented a valid **switch** statement). See §7.6: *Jump Statements* on page 108 for more information.

**Nested switch statements** You can nest **switch** statements. The **case** labels within a **switch** do not all have to be at the same scope level. You can enclose any **case** label within an arbitrary number of inner blocks. A **case** label is associated with the **switch** statement that immediately encloses it. A **case** label cannot be the target of a **goto** statement.

---

## 7.4.2 if Statement

The **if** statement allows conditional execution of statements and expressions in your program. This is the form of an **if** statement:

```

if ( arithmetic_expression )
    statement
[ else statement ]

```



When your program executes an **if** statement, *arithmetic\_expression* is evaluated, and if it is a non-zero value, the *statement* following the **if** is executed. If *arithmetic\_expression* evaluates to 0 (zero), control transfers to the **else** clause, if one is present. The *statement* following **else** or **if** can be another **if** statement.

*arithmetic\_expression* must be of arithmetic or pointer type, or a class type that can be unambiguously converted to a pointer or arithmetic type.

---

## 7.5 Iteration Statements

An *iteration statement* is a statement that causes itself and possibly a group of other statements to be executed repetitively. C and C++ iteration statements are **do**, **while**, and **for**.

---

### 7.5.1 **do** Statement

A **do** statement causes the statement that immediately follows it to execute repetitively, until the expression associated with **while** becomes 0 (zero). A **do** statement has the following form:

```
do statement while ( expression );
```

where the components have the following meanings:

*statement* Any valid C++ statement, except a declaration.

*expression* Any valid C++ expression whose result is an arithmetic type or pointer type, or a class type that can be converted to an arithmetic or pointer type.

*statement* is often a compound statement containing a list of other statements to be executed. *statement* is always executed at least once. If a **continue** statement appears in *statement*, control transfers to the evaluation of *expression*. *expression* is evaluated after each execution of *statement*. If *expression* contains side effects, those side effects complete execution before the next iteration of *statement*.

---

## 7.5.2 **while** Statement

The **while** statement allows repetitive execution of a *statement*, which may be a compound statement, until the *expression* associated with the **while** keyword reaches 0 (zero). This is the form of a **while** statement:

```
while ( expression ) statement
```

where the components have the following meanings:

- expression* Any valid C++ expression whose result is an arithmetic type or pointer type, or a class type that can be converted to an arithmetic or pointer type.
- statement* Must not be a declaration. C++ allows any other valid statement.
- expression* Must be evaluated before each iterative execution of *statement*. This means that *statement* might never execute at all, if *expression* evaluates to 0 (zero) the first time it is evaluated. If *expression* contains side effects, those side effects complete execution before the next iteration of *statement*.

---

## 7.5.3 **for** Statement

The **for** statement allows you to execute a *statement* repeatedly. A **for** loop consists of the **for** statement and the *statement* that follows it, called the **for** loop body. A **for** loop body is often a compound statement. Often, the **for** loop executes a specific, counted number of iterations. This is the form of a **for** statement:

```
for ( for_init [ cond_expr ] ; [ inc_expr ] )  
    statement
```

where the components have the following meanings:

- for\_init* Any valid C++ statement that initializes a data item. You can even declare a variable in this statement.
- If the statement is a declaration, any names you declare have a scope that extends to the end of the block enclosing the **for** loop. *for\_init* is typically an assignment that initializes a

variable, referred to as the **for** loop index. Because most statements end in a semicolon, a semicolon appears after the initialization.

*cond\_expr* Any valid C++ expression of a pointer or arithmetic type. The expression can also be of a class type, if an unambiguous conversion to pointer or arithmetic type exists for that class type. *cond\_expr* is usually a relational expression that compares the **for** loop index with a final constant value. The **for** loop iterations cease when evaluation of *cond\_expr* yields 0 (zero).

*inc\_expr* Any valid C++ expression. *inc\_expr* often changes the value of the **for** loop index. Often, *inc\_expr* is an expression that increments or decrements the **for** loop index by a constant value.

*statement* Any valid C++ statement except a declaration.

Both *cond\_expr* and *inc\_expr* are optional. If you omit *cond\_expr*, the **for** statement becomes equivalent to **while(1)**, and will execute forever, unless a jump statement appears in the **for** loop body.

*Flow of control in a for loop* When executing a **for** loop, *for\_init* executes first. This initializes the **for** loop index with a starting value. *cond\_expr* executes next.

If the result of *cond\_expr* is 0 (zero), execution of the **for** loop proceeds no further, and the **for** loop body is skipped. If the value of *cond\_expr* is non-zero, the body of the **for** loop executes.

After the body executes, *inc\_expr* is evaluated, followed by a re-evaluation of *cond\_expr*, again comparing *cond\_expr* to 0 (zero). This process continues until the condition *cond\_expr* == 0 is satisfied. Any side effects in *for\_init* or *inc\_expr* complete execution before *cond\_expr* begins execution.

*continue statement* During the execution of the **for** loop body, if a **continue** statement is encountered, control transfers beyond the end of the loop body to the evaluation of *inc\_expr*.

*break statement* A **break** statement in the loop body transfers execution to the next statement after the loop body, and terminates execution of the loop. See §7.6.1: *break Statement* on page 109.

*Labeled statement* You can use a labeled statement in a **for** loop body, and jump into the body without executing *for\_init* or *cond\_expr* first. However, such a jump will leave the **for** loop index uninitialized, and the behavior of the program is undefined.

## 7.6 Jump Statements

A *jump statement* is a statement that causes execution of a program to stop in one location and resume in another location. C++ defines four jump statements: **break**, **continue**, **return**, and **goto**.

### C++ Only

Jump statements automatically cause the destruction of objects declared in the scope of the jump statement, if the object is not also declared in the scope of the target. A jump statement cannot cause flow of control to enter a block and skip the construction of an object. For example:

```

struct s{
    s();
    ~s();
};
f(){
    s x1;           // Constructs x1
    {
        s x2;       // Constructs x2
        goto L1;    // Destroys x2
    }
L1: goto L2;        // Invalid: skips construction of x3
    {
        s x3;       // Constructs x3
L2: goto L3;        // Destroys x3
    }
L3: return;        // Destroys x1
    }

```

---

### 7.6.1 **break** Statement

The **break** statement is allowed only within an iteration statement (see §7.5: *Iteration Statements* on page 105) or a **switch** statement. This is the form of a **break** statement:

```
break ;
```

A **break** statement causes execution to transfer to the first statement following the immediately enclosing iteration statement, or to the next statement following the **switch** statement.

**break** is a convenient method for exiting a loop without the use of **goto**.

---

### 7.6.2 **continue** Statement

The **continue** statement provides a way to skip over the rest of an iteration of a loop. This is the form of a **continue** statement:

```
continue ;
```

When **continue** executes, execution transfers to the point just before the test for executing the next iteration of the loop. All remaining statements and expressions in the loop body between **continue** and the end of the loop are skipped.

---

### 7.6.3 **return** Statement

The **return** statement causes execution of the function containing the **return** statement to terminate. Execution resumes at the next statement or expression beyond the point where the function containing the **return** statement was called.

This is the form of a **return** statement:

```
return [ expression ] ;
```

A **return** statement is required to return a value, *expression*, unless the function that contains the **return** statement is a constructor, a destructor, or of type **void**. You cannot return a value for constructors, destructors, or **void** functions.

If necessary, *expression* is converted from its result type to the return type of the function, including possibly the construction of a temporary object.

A function that is not a constructor, destructor, or of type **void** must have a **return** statement that specifies a return value. It is not valid to execute to the end of a function without an explicit **return** statement.

---

#### 7.6.4 **goto** Statement

The **goto** statement transfers execution to a labeled statement (see §7.1: *Labeled Statements* on page 100). This is the form of a **goto** statement:

```
goto label;
```

*label* must be an identifier that corresponds to a defined label.

<b>Extension</b>
------------------

As an extension to ANSI Standard C and C++, High C++ allows you to specify a label in a **goto** statement where the label definition is in an outer function. If execution is currently at a **goto** statement in a nested function, you can specify a label in an outer, surrounding function as the target of the **goto**. See Chapter 8: *Functions* and Appendix B: *High C/C++ Extensions* for more information about nested functions.

# 8

## Functions

---

A *function* is a collection of expressions, statements, and data. A function declaration specifies:

- the name of the function
- the return data type, if any
- the parameter list

A function cannot be declared as **auto** or **register**. By default, functions are **extern**.

*Declaring a function*    The most frequent form of function declaration is the following:

```
decl_specifiers identifier ( argument_decl_list )  
                                [ cv_qualifier_list ]
```

This is just a form of a declaration (explained in §5.1: *Declarations* on page 48), where the declarator is a simple *identifier* followed by the parenthesized *argument\_decl\_list*. For example:

```
static int func ( )
```

Here you see no *cv\_qualifier\_list*, and the *argument\_decl\_list* is empty (see page 112 for the definition of an *argument\_decl\_list*). `func()` is a **static** function (**static** is a *storage\_class\_specifier*) returning an **int** (**int** is a *type\_specifier*).

The function declarator can be more complex. For example:

```
static int (* func ( ) )[]
```

Here `func()` returns a pointer to an array of **ints**. See §5.5: *Declarations and Precedence* on page 70 for how to read more complex declarators such as this. For the purpose of declaring a function, remember that the “innermost” declarator must be of the function declarator form: a *declared\_name* followed by a parenthesized *argument\_decl\_list*.

**C++ Only**

The optional *cv\_qualifier\_list* following the function’s right parenthesis pertains only to class-member functions and is explained in §9.5.3: *Qualified Member Functions* on page 136.

*Default return type* If you do not specify a type in the *decl\_specifiers* for a function, the is **int** function return data type defaults to **int**. If the function does not return any value, it must be declared type **void**. Any function that is not declared type **void** must return a value. See §5.4: *Data Types* on page 52 for more information about data-type declarations.

*Declaring arguments* An *argument\_decl\_list* can be empty or can be a list of declarations separated by commas, optionally followed by ... (ellipsis):

*argument\_decl\_list* is defined as follows:

```
argument_declaration [ , argument_declaration ] ...
                    [ [ , ] ... ]
```

An *argument\_declaration* is similar to the *declaration* in §5.1: *Declarations* on page 48:

```
decl_specifiers [ declarator | abstract_declarator ]
                [= expression ]
```

Here the difference is that *declarator* [*init\_declarator*] has been replaced. Its simplest form, *decl\_specifiers declarator*, is a subset of the *declaration* shown in §5.1: *Declarations* on page 48. But an argument can have a default value, specified by = *expression* — see §8.6.5: *Default Parameters* on page 121.

An *abstract\_declarator* is the same as a declarator, except that the *declared\_name* is omitted. Thus if:

```
*x[]
```

is a *declarator*, then:

```
* []
```

is the corresponding *abstract\_declarator*; *x* is omitted. Here is an example of a function declaration using a *declarator* for each argument:

```
int f ( int x[], int (*y)(), float z)
```

Here is the same declaration, using an *abstract\_declarator* for each argument:

```
int f ( int [], int (* )(), float )
```

The only difference between these two function declarations is that the former gives you the names of the arguments. When you define the function body,



be sure to specify the argument names if you want to access the arguments in the body!

#### C++ Only

The **const** and **volatile** keywords can be used after the parenthesis that closes the *argument\_decl\_list* only if the function is a non-**static** member function. See Chapter 9: *C++ Classes* for more information about **virtual** and member functions.

#### Defining a function

A function definition has two parts: the header and the body. The header is the same as the function declaration shown on page 111. The body can be defined with a given name and parameter list only once. A function body has the following form:

```
{ [declarations_and_statements] }
```

#### Function prototypes

You must declare a function before you can call it. If the body of a function is defined after a call to the function, a function prototype must appear in the source file before the call. A *function prototype* is a function declaration that does not define a function body. A function prototype ends with a semicolon.

See Chapter 6: *Expressions and Conversions* for more information about expressions, and Chapter 7: *Statements* for more information about statements.

---

## 8.1 Recursive Functions

Functions in C or C++ can be recursive, except for the function **main()**. A *recursive function* is one that calls itself, or is called while another call to the same function is still actively executing.

---

## 8.2 Nested Functions

#### Extension

As an extension to ANSI Standard C and C++, High C/C++ allows you to nest function definitions. For example:

```
#include <stdlib.h>
char outerfunc(){
    int i;
```

```

    int innerfunc(char c){ // This nested function is
        return atoi(c);    // contained inside another
    }                      // function.
    i = innerfunc('z');
}

```

*Static link for up-level access* The body of a nested function can access variables in the enclosing function. To make this possible, the nested function is passed a secret argument called a *static link*. Therefore the address of a nested function cannot be assigned to a pointer-to-function; the required static-link pointer would be lost.

See Appendix B: *High C/C++ Extensions* for more information about nested functions.

---

## 8.3 The Function `main`

This is the simplest possible complete C or C++ program:

```
main() { }
```

This program defines a function called `main()`, taking no arguments, which does nothing. Every C or C++ program must have a function called `main()`, which can be arbitrarily simple or complicated. Execution of every C or C++ program begins with the first statement or expression in `main()`. You cannot call `main()` directly from within your program, nor take its address.

*Command-line arguments* If your program expects to receive arguments from the command line when it is invoked, you can receive those arguments and process them by declaring `main()` as follows:

```

main(int argc, char *argv[]) {
    ...
}

```

*Parameters `argc` and `argv`* By convention, `main()` is called with two parameters. The first parameter is `argc`, which is the number of arguments that were passed to your program. The second parameter is `argv`, which is a pointer to an array of character pointers. These character pointers are the addresses where the actual character representations of the arguments from the command line reside. The first pointer in `argv`, `argv[0]`, points to the name of your program. `argv[1]` points to the first argument on the command line, if any, while `argv[2]` points to the next argument, and so on. `argv[argc]` is always 0 (zero).

## 8.4 inline Functions

The keyword **inline** is a hint to the High C/C++ compiler to substitute the body of a function declaration at each occurrence of a call to the function. It is only a hint, and the compiler might ignore it. Although High C/C++ can inline functions of arbitrary size and complexity, functions that exceed a user-specifiable size limit are not substituted. See the **High C/C++ Programmer's Guide** for more information about compiler options for inlining.

*Optimizing effect of inlining* Inlining is an optimization; inlining a function does not change the semantics of the function call in any way. When the compiler inlines a function, your program usually executes faster. This is because function-call overhead, such as saving and restoring registers, has been eliminated. Inlining can also make the program smaller when the function being inlined is so small that the code generated to issue a call to the function exceeds the size of the function-body code.

*Inlining versus macros* Inlining contrasts with the ANSI Standard C tradition of using a macro instead of a function call to gain program execution speed. If the body of the macro references a macro parameter more than once, undesirable side effects can occur when the argument is an expression containing side effects, such as `*p++`.

*Avoid multiple definitions of inline functions* An **inline** function must have exactly the same definition in every compilation in which it appears. If the **inline** function's definition is changed in two different source files, program behavior is undefined. *This constraint is not enforced by High C/C++*. For example, if you compile a source file using one definition of an **inline** function, where the definition of the **inline** function is in a header file, then modify the header file and compile another source file using the same function, the semantics of the **inline** function might have changed. Here is an example:

```
inline float sqr(float f) { return f*f; } // Square f.
...
float f1 = 3.7, g = sqr(f1);           // sqr is inlined.
```

```

struct s {
    float f;
    float sqr() { return f*f; }      // Square member f.
    // (Functions defined in a class definition
    // are inline by default.)
};
s x1 = {3.7}, g = x1.sqr();         // sqr is inlined.

```

---

## 8.5 friend Functions

The keyword **friend** declares a function to be a “friend” of a class. The keyword is valid only within a class declaration. See §10.4: *Friends* on page 159 for details and examples.

---

## 8.6 Function Parameters

*Formal and actual parameters* You can pass parameters to a function when you call it. A function’s *formal* parameter refers to an *argument\_declaration* in the function header where the function is declared or defined. An *actual* parameter is a value used when the function is called. Informally, the formal-parameter name in the *argument\_declaration* assumes the actual-parameter value during the execution of the body of the function.

*Formal parameters are auto* Formal parameters have storage category **auto**. Except for **register**, you cannot use storage-category keywords in the formal-parameter list.

### Example 8.1 struct astruct { float a, b };

---

```

extern double afunc( long xcoor, int ycoor,
    int (*p)(int), struct astruct x);
int inc( int i ){
    return i++ ;
}

```

```
main(){
    int x, y;
    struct astruct a;
    double result;
    x = 10; y = 20;
    result = afunc( x, y, &inc, a);
}
```

In Example 8.1, `xcoord`, `ycoord`, `p`, and `x` are formal parameters for the function `afunc()`. `i` is a formal parameter for the function `inc()`.

*Calling a function* You call a function when you use its name followed by parentheses enclosing an actual parameter list. The actual parameter list can be empty if the function you are calling has no formal parameters. An empty actual parameter list is indicated by a parenthesis pair `()`. In Example 8.1, `afunc()` is called passing the actual parameters `x`, `y`, `&inc`, and `a`. When this code executes, the next source statement that executes following the call to `afunc()` will be the first statement of `afunc()`.

*Parameter order and type compatibility* Actual parameters are associated with formal parameters in the order in which they appear in the parameter lists. The data types of an actual parameter and the corresponding formal parameter must be valid according to the rules of assignment compatibility. See §6.4.1: *Assignment* on page 79 for more information about assignments. If the actual and formal parameters are of compatible but not identical data types, the actual parameter is converted to the data type of the formal parameter before it is passed. In Example 8.1, `xcoord` and `x` are of different but compatible types. When `x` is passed to `afunc()`, `x` is widened to `long`.

---

## 8.6.1 Passing Parameters by Value and by Reference

*Passing by value is the default* When you pass an actual parameter to a function, the actual parameter is evaluated and the resulting value is passed to the function. This is called passing parameters *by value*. Inside a function, if you assign a value to one of the function's formal parameters, the value of the actual parameter is not changed. In Example 8.1 on page 116, if `afunc()` changes `xcoord`, then when `afunc()` finishes executing and returns to `main()`, the value of `x` is unaffected. By default, parameters are passed by value in C and C++.

*Passing an array* An actual parameter that is an unsubscripted array name is passed as a pointer to the first element of the array. This means you cannot pass an array by value. See §5.4.5.3: *Arrays* on page 61 for more information about arrays.

Note the actual parameter `&inc` in Example 8.1 on page 116. The `&` operator means “address of”. `inc()` is a function; `p`, the corresponding formal parameter, is a pointer to a function. The address of `inc()` is passed to `afunc()` and assigned to `p`. When the actual parameter is an address, the formal parameter must be a pointer. In the example, the `&` operator is not necessary for a function parameter — it is supplied automatically if it is not explicitly mentioned. However, `&` is implicit only for function parameters.

*Passing an address* When you pass the address of a function or data item, you are passing the value of a pointer to function or data item. If you are used to programming in other languages, such as FORTRAN or Pascal, you can think of this as passing the parameter *by reference*.

---

## 8.6.2 Reference Parameters

### C++ Only

In C++ you can specify a *reference parameter*. This allows you to pass a parameter to a function, assign a value to it in the called function, and have the actual parameter’s value changed. The change occurs immediately.

```
void func(double& radius) {
    radius += 3.1416;
}
main(){
    double d=0.0;
    func(d);
    cout << d;
}
```

In this example, `radius` is declared as a reference argument of `func()`. The call to `func()` causes `d` to have the value `3.1416`.

*Reference variables* Although the reference mechanism is most widely used in parameter passing, it is generalized in C++ to include arbitrary reference variables; that is, you do not need to pass a parameter to make a reference to a variable. See §5.4.5.2: *References* on page 60.

You cannot dereference a pointer to a function and assign a value through it. For a data item, if you dereference the pointer (*\*p* in Example 8.1 on page 116) and assign a value through it, the value of the actual parameter is changed.

---

### 8.6.3 Parameter Evaluation Order

*Side effects* A *side effect* is a change to a variable as a by-product of an expression evaluation. For example, a function call or an increment can change a variable.

The order in which actual parameters are evaluated is implementation dependent. Avoid writing code where the value of an actual parameter depends on the order of evaluation. The order of parameter evaluation can even be different on different architectures! For example:

```
int k=1;
int g(int& myint){
    return myint++;
}
int func (int i, int j, int k) {
    return i+j+k;
}
main(){
    func(k, g(k), k++);
}
```

In this example, the three arguments can be evaluated in any order, and the last two arguments affect the value of *k*. In both cases an increment affects the value, and the increment is hidden within a function in one case. Thus the value of the first argument depends upon when the last two arguments are evaluated.

---

### 8.6.4 Variable-Length Parameter Lists

When you specify a formal-parameter list, you can allow a function to have a variable number of parameters by using the notation “...” at the end of the list of formal parameters. For example:

```
void printints(int last, ...);
```

This function prototype declares a function `printints()` that prints one or more integers. `printints()` has one fixed parameter followed by an unknown number of other parameters. Within `printints()` you can access additional parameters beyond the fixed parameter by means of the `va_start()`, `va_arg()`, and `va_end()` macros defined in `<stdarg.h>`.

### Example 8.2 Accessing a Variable-Length Parameter List

---

```
#include <stdarg.h>
#include <stdio.h>
void printints(int Last,...){
    int i;
    va_list ap;
    va_start(ap,Last);
    for (i=0; i<Last; i++)
        printf("%d",va_arg(ap,int));
    va_end(ap);
}
main(){
    // Prints 0, 1, 2, and 3
    printints(4,0,1,2,3);
}
```

---

In Example 8.2, an integer parameter `Last` is passed in to indicate the number of parameters that follow. A variable `ap` is declared as type `va_list`. The `va_list` type comes from `<stdarg.h>`. The `va_start()` macro assigns the address of the beginning of the variable-length parameter list to `ap`. The `va_arg()` macro then references each parameter in the list of unknown parameters. Each time `va_arg()` is referenced, it returns the next parameter in the list of unknown parameters.

A typical application of variable-length parameters is in constructing the `printf()` function, whose declaration is as follows:

```
int printf(char *fmt, ...);
```



---

## 8.6.5 Default Parameters

**C++ Only**

You can declare a function that has default parameters. For example:

```
int myfunct( int a, char b, double d = 5.0 );
```

The last parameter in the list for `myfunct()` is of type **double**. Its default value is `5.0`; this value is used if a call to `myfunct()` omits the last actual parameter. For example:

```
int i;  
i = myfunct(3, 'a'); // Passes 5.0 as third argument
```

You can still call `myfunct()` with three arguments to override the given default:

```
int i;  
i = myfunct(3, 'a', 6.0); // Passes 6.0  
                          // as third argument
```

If a default parameter is given, all parameters following that default parameter must also have defaults:

```
void func1( int a,      int b = 2, int c); // Error  
void func2( int a,      int b = 2, int c=3); // OK  
void func2( int a = 1, int b = 2, int c=3); // OK
```

Notice that both declarations of `func2()` are allowed. After the first declaration, `func2()` has two default parameters and could be called with a single value. After the second declaration, a third default is supplied, for the first parameter. Thus, default parameters are cumulative.

A default parameter does not affect a function's definition. It affects only calls to the function. The same function could be declared in different ways in different source modules, with different default parameters.



# 9

## C++ Classes

---

*Code and data in the same structure*

This chapter describes C++ classes. The class is the foundation of C++’s object-oriented programming model. Understanding the properties of classes is crucial to effective use of C++. With classes you can define new types where code and data are present in the same data structure and access to data is restricted to a specific set of access functions.

*Objects*

In discussing classes, this chapter uses the term *object* to refer to an area of storage occupied by a particular allocation of a class or class member.

*Encapsulating data by restricting access*

By restricting access to data within a class you *encapsulate* the data, as though putting a protective covering around it to prevent unauthorized access. In a **struct** or **union**, any function in the proper scope can manipulate the **struct** or **union** members. But in a class, by default the class members cannot be read or written to except by other members of the same class.

*Classes extend the possibilities of structures*

C programmers need not be frightened of classes; a subset of classes encompasses exactly ANSI Standard C **structs**. Think of classes as “super-**structs**”, **structs** with many more features and possibilities than plain C **structs**. The power of classes is essential to the object-oriented design of C++. But in C++ you are not forced into programming with all of that power; you can stick to the plain C subset if you wish. In fact, the **class** keyword is really largely unnecessary — you can use **structs** and **unions** to perform the same functions as classes in C++ — only the default access rules for members are different.

---

## 9.1 Class Declarations and Definitions

This is the basic syntax for a *class\_definition* (first presented in Chapter 5: *Declarations and Definitions*):

```
[_Packed | _Unpacked] {class | struct | union}  
  [identifier] [base_list]  
  { {member_declaration | access_specifier : ... } }
```

Defining a **class** This is the form of a **class** definition:

```
class_head { [member_list] }
```

where the components have the following meanings:

**member\_list** is:

```
{ member_declaration | access_specifier : } ...
```

**member\_declaration** is one of the following:

```
[ decl_specifiers ] [member_declarator_list] ;  
function_definition [ ; ]  
qualified_name ;
```

**member\_declarator\_list** is:

```
member_declarator [ , member_declarator ] ...
```

**member\_declarator** is one of the following:

```
declarator [ pure_specifier ]  
[ identifier ] : constant_expression
```

**class\_head** is one of the following:

```
class_key [ identifier ] [ base_spec ]  
class_key class_name [ base_spec ]
```

**class\_key** is:

```
{ struct | union | class }
```

**base\_spec** is:

```
: base_list
```

**base\_list** is:

```
base_specifier [ , base_specifier ] ...
```

**base\_specifier** is one of the following:

```
complete_class_name  
virtual [ access_specifier ] complete_class_name  
access_specifier [ virtual ] complete_class_name
```

**class\_name** is an identifier that is the name of a previously declared class.

**access\_specifier** is an access-control level:

```
{private | protected | public}
```

*complete\_class\_name* is:

```
[::] qualified_class_name
```

*qualified\_class\_name* is:

```
class_name [:: qualified_class_name]
```

*pure\_specifier* is = 0

**class** versus **struct** In a *class\_definition*, **class** and **struct** can be used interchangeably, except that the use of **class** in a definition implies different default access rules (see §10.1: Access on page 151). Example 9.1 shows many facets of a class definition.

### Example 9.1

---

```
class A : public B, virtual C {
    // B is a public base class.
    // C is a virtual base class.
    // First, some declarations that are also
    // allowed in structs and unions in ANSI C:
    int x;           // A normal member
    float f[10];     // Another normal member
    unsigned j:3;    // A bit field
    enum color {red, white, blue}; // An enumeration
    // The rest of these are allowable only in C++:
    void f();        // A function member
    public:          // An access specifier
    int sqr() {      // A function member, together
        return x*x; // with its definition
    }
    ; ; ; ; ;       // Empty member_declarations
    typedef int *PI; // A typedef member
    static char c;   // A static member
    A *next;         // Use of the class name A
                    // declaration
```

```

class A *next2;    // The word "class" is
                  // redundant here, but
                  // allowed.
private:          // Another access qualifier
B::g;             // Adjusting access of g,
                  // inherited from B
// The only allowable case of an initializer:
virtual void func() = 0;

// Some *** invalid *** declarations:
auto local_var;   // Cannot use auto
int_array[10];    // Data type is omitted;
                  // "int int_array[10];"
                  // would be correct.
void func2() = 1; // 1 is not valid. Also,
                  // "virtual" is omitted.
A sub_a;          // Cannot contain an instance
                  // of an incomplete type
};

```

---

- Classes define new types* Every class definition defines a new type different from all other types. The type is incomplete until the ending curly brace (}) of the definition is seen; it is then made “complete”.
- Empty member declarations* It is possible for a *member\_declaration* to produce nothing, because all parts of it are optional, so *;;;;;* can appear within a class definition.
- Scope of class members* A class definition introduces a new scope at the occurrence of {, and the scope closes at the corresponding }. This means that names declared as members are local to the class definition. The class definition corresponds to the traditional ANSI Standard C **struct** or **union** declaration. But in C++, you can also declare and optionally define functions within the class that allow access to private names declared in the class definition. You can also adjust access to names inherited from base classes and specify so-called *static members* of a class (see §9.4.3: *static Members* on page 131).

## 9.2 Local Classes

You can declare a class within a block, such as a function definition. Such a class is called a *local class*. The name of a local class is in the scope of its enclosing block, and cannot be “seen” outside the function. If you declare a local class, its members can use only type names, **static** variables, **extern** variables, functions, and enumerators from the enclosing scope level. For instance, accessing **auto** variables declared in the enclosing function is invalid:

```
void func() {
    int i;
    enum a {b,c};
    static int j;
    struct s {
        int a[i];           // Use of i not valid
        enum a x;           // OK
        void s_member() {
            j = 3;           // OK
            x = c;           // OK
            func();          // OK
        }
    };
}
```

## 9.3 Nested Classes

A class declared inside another class is a *nested class*. The class name of a nested class is local to its enclosing class. You can use only type names, static members, and enumerators from the enclosing class in a nested class declaration, unless you explicitly use pointers or access operators.

- Member functions of a nested class do not have any special access to members of the enclosing class.
- Member functions of the enclosing class have no special access to members of the nested class.

## 9.4 Class Members

A *class member* is a declared type definition, variable, or function whose scope is restricted to its containing class. Access to a particular class member can be restricted via access qualifiers (see §10.1: *Access* on page 151).

*Rules for declaring simple members*

The following rules apply to member declarations:

- In a member declaration you cannot use **auto**, **extern**, or **register**
- You can omit the data type in a member declaration; if you do, **int** is implied.
- You can use the `= 0` syntax in a *member\_declarator* only if the name declared is a **virtual** function type. Doing this declares a *pure virtual function*. See §9.7: *Virtual Functions* on page 141 for more information about **virtual** functions.
- The type of a member of a class type must be complete. The class type is incomplete until the ending curly brace `}` of the definition is seen; it is then made “complete”. This prevents a class from containing an occurrence of itself. For example:

```
class s {
    s x;           // This is not valid.
};
```

This declaration results in an “infinite” definition: **s** contains an **s** which contains an **s** which contains an **s**, and so on. However, a class can contain a pointer to itself, as shown in the following example:

```
class boy {
    int height;
    int weight;
    enum color { blue, brown, black, green, hazel };
    boy *next; // A pointer to an object of this
               // class type
}
```

### 9.4.1 Simple Members

A simple class member is the same as a member in a **struct** or a **union** in ANSI Standard C.



*Rules for declaring simple members*

The following rules apply to simple member declarations:

- Specifiers exclude the storage-category qualifiers and function specifiers.
- `= 0` (in *member\_declarators*) is not used.
- The member is not of a function type.
- The declarator is required in the *member\_declaration*.
- When the declarator is omitted, a useful member declaration occurs only when the specifiers specify an enumeration or class definition.

*Storage for simple members*

If a class type is declared using **struct** or **class**, each object of the class type contains a separate copy of all simple members, and the **sizeof** value of the class type is big enough to accommodate all the members.

If a class type is declared using **union**, the **sizeof** value of the class type is large enough to accommodate the largest member, and each member shares storage with the other members. In particular, the address of each non-bit-field member is the same as any other non-bit-field member. You can think of a union as a class whose member offsets are all 0 (zero), and that is big enough to hold any of its members.

**Example 9.2**


---

```

struct A {
    int x;
    float f[10];
    unsigned j:3;
};
main() {
    A a1, a2, a3, *a4 = &a3;
    a1.x = 3;      a2.x = 4;      a4->x = 5;
    a1.f[4] = 3.3; a2.f[6] = 4.4; a4->f[7] = 5.5;
    a1.j = 1;      a2.j = 2;      a3->j = 3;
}

```

---

In Example 9.2 the declaration of a **struct** is the same as in ANSI Standard C. If instead of **struct A** we declared **union A**, the address of **x** and of **f** would be the same; that is, **x** and **f** would overlap.

*Declaring bit fields* If the identifier is omitted in a bit-field declaration, no storage is allocated, but the effect is to direct placement of subsequent members at the beginning of the next bit-field container.

When the declarator is omitted, a useful member declaration occurs only when the specifiers specify an enumeration or class definition. For example:

```
struct A {
    int;                // Not useful, nothing happens
    enum color {red, white, blue}; // An enumeration
                                // definition
    struct B {           // A nested class
        int x;
    };
};
```

This example is not much different from ANSI Standard C, either. However, the difference between C and C++ here is that in C++, the type names `color` and `B` and the enumeration literals `red`, `white`, and `blue` are local to `A`. In ANSI Standard C those names are visible outside the `struct`. For example:

```
// ANSI C usage.
struct B x;
enum color c = red;
```

*Accessing names with the scope resolution operator* In C++, these names are not visible. However, they can be accessed via the C++ scope resolution operator (`::`). The following example shows how to access these names.

```
A::B x;                // A::B means name B within A.
A::color c = A::red;
```

See §5.4.2: *Scope Access* on page 55 for information about `::`. You can make type names as visible as with ANSI Standard C by using `typedefs`:

```
typedef A::B B;          // Defines B to be the
                        // same type as A::B
typedef A::color color; // Same for color
```

*Class objects in C versus C++* Each object of a class type has a copy of each of the simple members, distinct from the copies in another object. But this is no different from the C language, and in fact C++ adds nothing to C in this regard. C++ adds to C by increasing the kinds of things you can do with a class object or its members, and by allowing declarations of members that are shared across all objects of a class.

---

### 9.4.2 `typedef` Members

When you use `typedef` in a member declaration, the effect is the same as a `typedef` declaration elsewhere, except that the declaration is local to the class:

```
class A {
    typedef float F;
    typedef A * (*X)[3];
};
A:F f;      // f is of type float.
A:X d;      // d is of type
             // pointer-to-array-of-pointers-to-A.
```

---

### 9.4.3 `static` Members

When you declare a `class` member `static`, and the member you are declaring is not a bit field or a function, the declared name denotes exactly one object, shared among all objects of the class. The member declaration does not allocate storage for the `static` object; a definition is required elsewhere. Because the `static` object is a member, you can access it with a class object followed by the standard `.` or `->` operator. But you can also access the `static` object without a class object via `::`. For example:

```
class A {
    static int count;
};
int A::count = 0; // Required definition
main () {
    A a1, a2;
    a1.count++;
    a2.count++;
    A::count++;
    // Now A::count has the value 3
}
```

The difference between placing `count` within the class and making it global is that within the class `count` is class specific. Other classes can have the same member declaration, but it denotes a different `count` variable in each case.

For example:

```
class A {
    static int count;
};
class B {
    static int count;
};
int A::count = 0, B::count = 1;
// There are two count variables here.
```

**static** bit-field members are not allowed. **static** function members are allowed (see §9.5.1: *static Member Functions* on page 133). The initializer of a static data member is “in the scope of the class” — that is, the names of all class members are known and can be referred to without the use of `.`, `->`, or `::` operators. For example:

```
class A {
    enum color {red, white, blue};
    color C;
};
A::color A::C = red; // red means the same as A::red
```

---

## 9.5 Member Functions

*Member functions are methods* A class member declaration can be a function declaration or a function definition. Such a member is called a *member function*. Other object-oriented languages refer to such a member as a *method*. If you do not supply a function body in the declaration, you must supply the function body for the member function elsewhere (if you reference the function).

Each member function is shared among all objects of a particular class type. It can be accessed via `.`, `->`, or `::`.

**inline** is implied for a member function whose definition is supplied within a class definition. You cannot prevent a function from being **inline** when you supply the function body in the class definition. See §8.4: *inline Functions* on page 115.

### 9.5.1 static Member Functions

A *static member function* is a function whose declaration and possibly definition occur within a class declaration, and the declaration contains the storage category **static**. Here is an example of a static member function:

```
class A {
public:
    static int count;
    static void inc_count();
};

int A::count = 0;
void A::inc_count() { A::count++; }

main () {
    A a1,a2;
    a1.inc_count();
    a2.inc_count();
    A::inc_count();
    // Now A::count has the value 3.
}
```

Unlike **static class** data objects, the function definition can be provided within the **class** definition itself. For example:

```
struct A {
    static int count;
    // Declaration and definition of inc_count:
    static void inc_count() { A::count++; }
};

int A::count = 0;
```

**inline functions** A function definition given within a class is automatically **inline**; see §8.4: *inline Functions* on page 115.

*Scope of member functions* A member function name, its parameter list, and any declarations local to the member function are in the scope of the enclosing class:

```
struct A {
    static void inc_count() { count++; }
                                // ^^^^^ A:: not needed
    static int count;
};
```

Even so, a **static** member function cannot make a simple reference to a non-**static** data member, because there is no object containing that data member. Were `count` non-**static**, the reference `count++` within `inc_count()` would not be valid.

---

## 9.5.2 Non-**static** Member Functions

Non-**static** member functions are much more common in C++ than **static** member functions, and are an important element of the object-oriented philosophy of C++. A non-**static** member function differs from a **static** member function in that a non-**static** member function is always supplied an object of the class in which the function was declared.

*Calling a non-**static** member function*

To call a non-**static** member function, you must have constructed an object of its class type. When the function is called, the address of that object is passed to the member function as a hidden argument. When that member function refers to the simple members of the class, it is referring to those members within the hidden object. For example:

```
#include <iostream.h>
struct string {
    char A[512];
    void concat(const char *s);
};
void string::concat(const char *s)
{ _strncat(A, s, sizeof A); }
main () {
    string s;
    s.A[0] = 0;
    s.concat("Hello, ");
    s.concat("world.\n");
    cout << s.A;
}
```

This is a "Hello, world" program. Within `concat()` a reference to `A` needs no `.` or `->` or `::` operator, and it refers to the `A` belonging to the hidden-argument class object. Whenever you call a non-**static** member function with the form `X.Y()`, the function `Y` is passed `&X` as the hidden argument; similarly, in `X->Y()`, `Y` is passed `X` (which is already an address).

*Hidden argument* **this** A pointer to the class object is passed in as a hidden first argument, and is named **this**. It is as if we had written the following C++ program:

```
#include <iostream.h>
struct string {
    char A[512];
};
void concat(string *this, const char *s) {
    _strncat(this->A, s, sizeof this->A);
}
main () {
    string s;
    s.A[0] = 0;
    concat(&s, "Hello, ");
    concat(&s, "world.\n");
    cout << s.A;
}
```

Although not as elegant as the member-function syntax, this is the way C programmers have been writing C programs with some object orientation for years. C++ gives you a syntactic boost.

*Dereferencing pointers with this* When you invoke a member function of a particular class, each reference to a member of that class within the function has a pointer dereference for the class object automatically supplied. The automatic pointer dereference is equivalent to **this->class\_member** for any class member. You can also use the **this** argument directly yourself:

```
void string::concat(const char *s) {
    _strncat(this->A, s, sizeof A);
}
```

With a little renaming, the following example shows a case where you must use **this** directly:

```
class string {
    char A[512];
    void concat(const char *A);
    void string();
};
void string::concat(const char *A) {
    _strncat(this->A, A, sizeof this->A);
}
```

In this example, `A` was purposely named as an argument to `concat()` to “hide” the class member `A`. Remember that for member functions, the parameter list of the member function is in the scope of the class. In the example, however, the parameter list (`const char *A`) essentially overrides the visible member `char A[512]`, so `this` is necessary to access `char A[512]`. For a member function of class `string`, the type of the `this` parameter is `*string`.

### 9.5.3 Qualified Member Functions

Non-`static` member functions can be qualified with `const` or `volatile`; the qualification appears after the right parenthesis in the function declarator (see Chapter 8: *Functions*).

The purpose of the qualifiers is to modify the type of the `this` hidden parameter. Because there is no place where `this` is explicitly declared, we cannot use the normal C++ syntax to specify its type.

For a member function `F` of some class `S`, the type of `this` is `S* const` — that is, constant pointer to `S` (you cannot modify `this`). But if the function is qualified, the type of `this` becomes `cv_qualifier_list S * const` — that is, the `cv_qualifier_list` modifies the type of the object to which `this` points. For example:

```
struct S {                                // Type of this:
    void f();                             // S* const
    void f() const;                       // const S* const
    void f() volatile;                   // volatile S* const
    void f() const volatile;             // const volatile S* const
    int x;
};
```

A function qualified with `const` essentially promises not to modify the object to which `this` points. For example:

```
void S::f() const { x = 1; }             // Error.
```

Here `x` is really `this->x`, and because `this`'s type is `const S * const`, you cannot modify the `S` object.

A function qualified with `volatile` recognizes that the object for which it is being called can be modified by forces unknown to the function, so references



to the object's contents are not optimized. See §5.4.6.2: *Type Qualifier volatile* on page 68 for more information about **volatile**.

Unless a function is qualified with **const**, it cannot be called with **const** objects; this is a normal consequence of function-call rules that prohibit passing a pointer of type **const T \*** where the actual parameter is of type **T \***. Similarly, unless a function is qualified with **volatile**, it cannot be called with **volatile** objects. To get around this restriction you would have to use a cast, although you should do this only when you know the called function will not modify the object (in the case of **const**):

```
struct s { void f(); };
void g() {
    const s x;
    x.f();           // Error
    ((s&)x).f();     // OK but dangerous: casting x to
                    // non-const s& before call
}
```

We recommend you use **const** for any member function that does not modify the object for which it is called. Such a member function can be called on behalf of both **const** and non-**const** objects.

Constructors and destructors are an exception: they cannot be declared **const** or **volatile**, but can be called on behalf of **const** or **volatile** objects.

---

## 9.6 Derived and Base Classes

*A derived class inherits information from a base class* An important part of object-oriented programming with C++ is the ability for a class to use information and attributes from other classes. A class **B** is a *base class* for another class **C** if **B** is referenced in the base list of **C** or any of **C**'s bases. A class that uses information from another class in its definition is a *derived class*. The information that a derived class uses from a base class is *inherited*.

Each *base\_spec* in a *base\_list* (see §9.1: *Class Declarations and Definitions* on page 123) specifies a base for a class **X**; each base is simply a previ-

ously defined class. **X** is said to be “derived from” the bases, and those bases are “bases of **X**”.

- If **virtual** is included in the *base\_spec*, the specified base is a *virtual base* of **X**.
- If an *access\_specifier* is given in a *base\_spec*, the base is a public, private, or protected base of **X**.
- In the absence of an *access\_specifier*, if a *class\_definition* uses the word **class**, each base is assumed to be a private base; if the word **struct** is used, each base is assumed to be a public base; if the word **union** is used, there is no issue, because unions cannot have base classes.

*Single and multiple inheritance* The term *multiple inheritance* means a derived class inherits information from more than one base class. *Single inheritance* means that a derived class has exactly one base class. A derived class and all of its base classes and their base classes are a *class hierarchy*. The derived class can also override virtual member functions of its base classes. See §9.7: *Virtual Functions* on page 141 for more information about virtual functions.

ou declare a class to be a base class as in Example 9.3 shows how to declare a base class.

### Example 9.3

---

```
enum color { red, green, blue, black, white };
class shape {
public:
    int x, y;           // Screen coordinates of center
    color background;   // Screen background color
    color shapeoutline; // Color to make shape outline
}
class circle : shape {
    int radius;
}
```

---

In Example 9.3, the **class** *shape* is a *private base class* for the **class** *circle*.

Notice that `circle` inherits from `shape` some characteristics that would apply to any shape displayed on a color screen. You could use the class members in `shape` as the basis of another class called `rectangle`. In the example, the `circle` class has *single inheritance*, because only one base class is specified for it. If more than one class name followed `class circle :`, then `circle` would have *multiple inheritance*.

The members of the base classes are said to be *inherited* by the derived class. This means they can be used after the `.`, `->`, or `::` operators just as if they were members of the derived class. Such use is not permitted if it is ambiguous — that is, if there is more than one possible member to select.

Example 9.4 shows inheritance of members from a base class.

### Example 9.4

---

```
struct A {
    enum color { red, white, blue };
    color C;
    // Print member C in some fashion.
    void print_C() {
        . . .
    }
};
struct B: A {
    color C2;
    int blue;
};
. . .
B b;
if (b.C == b.C2) b.print_C();
B::color C3 = B::red;
```

---

In Example 9.4, `B`:

- inherits `color`, `red`, `white`, `C`, and `print_C` from `A`
- declares an additional member `C2`
- overrides the name `blue` in `A`

Overriding the name `blue` means that a mention of `blue` in the context of class `B` yields `B::blue`, not the inherited name `A::blue`; the inherited name is effectively hidden. Therefore, `b.blue` refers to `B`'s `blue`.

The expressions `b.C`, `B::color`, and `B::red` in Example 9.4 on page 139 refer to the inherited names.

In `struct s2:s1 ...`, `s1` is a *direct* base of `s2`. Add `struct s3:s2...`, and `s1` is an *indirect* base of `s3`.

*Access control* Use of inherited members is also governed by access control; see Chapter 10: *Access Control in C++*.

A derived class can be thought of as a structure containing its own members, plus storage allocated for direct bases of the class. C++ allows access to the base's fields without explicitly naming the bases. Example 9.5 is written as if the language supplied no base classes.

### Example 9.5

---

```

struct A {
    enum color { red, white, blue };
    color C;
    // Print member hue in some fashion.
    void print_C() {
        . . .
    }
};

struct B {
    struct A A_base;    // Fake out a base.
    A::color C2;
    int blue;
};

. . .
B b;
if (b.A_base.C == b.C2) b.A_base.print_C();
A::color C3 = A::red;

```

---

The difference between Example 9.5 on page 140 and Example 9.4 on page 139 is that in some places in Example 9.5, `A::` replaces `B::`, `A::` is inserted,

or `A_base` is accessed to get to the members of the simulated inherited class. This is close to the way inheritance is actually implemented.

*Hidden argument* `this` When a member function inherited from a base class is called on behalf of a derived class object, the `this` pointer passed is adjusted to point to the base class sub-object within the derived class.

---

## 9.7 Virtual Functions

C++ allows you to declare functions in base classes that act as “place holders” for functions with the same names in derived classes. When you declare a function in a base class that has the `virtual` qualifier, and then declare a function with the same name and parameter types in a class derived from that base, the function in the derived class “overrides” the `virtual` function in the base class. You can use `virtual` for any non-static class member function.

The semantics of a virtual function are as follows:

- If a `virtual` function `f` in a base class `B` is overridden in a derived class `D`, any call to `B::f` on behalf of a `B` object contained within a `D` object instead calls `D::f` on behalf of the `D` object.
- A function `D::f1` overrides a virtual function `B::f2` if:
  - `f1` and `f2` are the same name and have the same type and number of parameters; for example, `f1 = f2 = func`.
  - `f1` and `f2` are both destructors; for example, `f1 = ~B` and `f2 = ~D`. See §11.2: *Destructors* on page 176 for more information.
  - `f1` and `f2` are conversion functions with the same name; for example, `f1 = f2 = operator int`.

Example 9.6 illustrates a use of virtual functions.

### Example 9.6

---

```
#include <stdio.h>
struct point { int X, Y;
    point(int x, int y): X(x), Y(y) {}
};
```

```

struct shape {
    virtual void draw() {
        printf("Nothing interesting here.\n");
    }
};

static void draw_many(shape *A[], int cnt) {
    for (int i = 0; i < cnt; i++) A[i]->draw();
};

struct square : shape {
    point upper_left, lower_right;
    void draw() { // Code to draw a square
        printf("I am drawing a square!\n\t"
            "upper_left=(%d,%d), lower_right=(%d,%d)\n",
            upper_left.X, upper_left.Y,
            lower_right.X, lower_right.Y);
    }
    square(point UL, point LR) : upper_left(UL),
                                lower_right(LR) {}
};

struct circle : shape {
    point center; double radius;
    void draw() { // Code to draw a circle
        printf("I am drawing a circle!\n\t"
            "center=(%d,%d), radius=%f\n",
            center.X, center.Y, radius);
    }
    circle(point C, double R) : center(C), radius(R) {}
};

struct line : shape {
    point left, right;
    void draw() { // Code to draw a line
        printf("I am drawing a line!\n\t"
            "left=(%d,%d), right=(%d,%d)\n",
            left.X, left.Y, right.X, right.Y);
    }
    line(point L, point R) : left(L), right(R) {}
};

```

```

main () {
    circle C(point(1,3), 4);
    square S(point(1,2), point(3,4));
    line   L(point(3,3), point(4,5));
    shape *A[] = {&C, &S, &L};
    draw_many(A,3);
}

```

*Constructors* To make Example 9.6 on page 141 an actual runnable program, we had to provide constructors. A *constructor* is a member function whose name is the same as that of the class. In Example 9.6, `point(1, 3)` constructs a `point` object whose `x` and `y` values are 1 (one) and 3 respectively; `circle C(point(1, 3), 4)` constructs a `circle` whose center and radius values are `point(1, 3)` and 4, respectively; and so forth. See §11.1: *Constructors* on page 165 for more information about constructors.

Example 9.6 shows a basic `shape` class with a **virtual** `draw()` function that does not do anything realistic; it is intended to be overridden by a `draw()` function in a derived class. Nonetheless, we can still define a `draw_many()` function that draws many shapes, calling `shape::draw()` on each one. The three derived classes define three realistic shapes: a square, circle, and line; each class provides an overriding draw function that can draw that specific shape.

When `draw_many()` is passed three `shape` pointers (`&C`, `&S`, `&L` get converted to a pointer to the base class `shape`), the call to `draw()` within `draw_many()` invokes the functions in `C`, `S`, and `L` instead, and the **this** pointer for those three calls is adjusted back up to the derived class from the base class. This makes it possible to write `draw_many()` without having to know all possible classes that might be derived from `shape`. Operationally here is what happens:

- `A[0]` is assigned the pointer to the `shape` base within `C`.
- `A[1]` is assigned the pointer to the `shape` base within `S`.
- `A[2]` is assigned the pointer to the `shape` base within `L`.

Within `draw_many()`:

`A[0]->draw()` adjusts `A[0]` to pointer to `C`, then calls `circle::draw()` on behalf of `C` (passing pointer to `C` as `this`).

`A[1]->draw()` adjusts `A[1]` to pointer to `S`, then calls `square::draw()` on behalf of `S` (passing pointer to `S` as `this`).

`A[2]->draw()` adjusts `A[2]` to pointer to `L`, then calls `line::draw()` on behalf of `L` (passing pointer to `L` as `this`).

If you run this program, you get the output:

```
I am drawing a circle!
  center=(1,3), radius=4.000000
I am drawing a square!
  upper_left=(1,2), lower_right=(3,4)
I am drawing a line!
  left=(3,3), right=(4,5)
```

How does `A[i]->draw()` “know” the pointer adjustment and the right function to call? The answer is that each object with any virtual member functions contains a pointer to a virtual-function table (see §9.7: *Virtual Functions* on page 141). That table is referenced whenever a call to a virtual function `f()` is made. The entry in that table for `f()` records the pointer adjustment and which function to call. The table has as many entries as there are virtual functions in a class; each function is assigned a distinct, compile-time-known location in that table. The pointer to that table is installed in the object when the object is created at run time (see §11.1: *Constructors* on page 165).

In Example 9.6 on page 141, you can think of `A[0]->draw()` as implemented by:

```
vp = &A[0]->vftab[draw_index];
vp->func(A[0]+vp->adjust);
```

where `vp` is a pointer to a virtual-function-table entry, and `draw_index` is the compile-time-known location for the virtual draw function within the table. `vp->adjust` is the value that converts the pointer to the base shape within a `C` to the pointer to the `C` object. The declaration of the `C`, `S`, or `L` object installs the appropriate virtual-function table within the `shape` sub-object of `C`, `S`, or `L`.



*Virtual calls and polymorphism* The use of the virtual-function table in a call to a virtual function is often referred to as the *virtual call mechanism*. The virtual call mechanism provides C++ with *polymorphism*: a function call can invoke one of a set of different functions, depending on the type of the object on behalf of which the function is called. Here, if the type is a `circle`, the circle's `draw()` function is called; if the type is a `square`, the square's `draw()` function is called; and so on.

*Virtual-function tables* The *set of virtual-function tables for a class X* is the collection of tables designed to override any virtual functions in any base class(es) of `X`, and to remap virtual functions in `X` to themselves. These tables are plugged into the virtual-function-table pointers for `X` and all of its bases. There may be a different virtual-function table for `X` and for each of its bases (although there are compiler optimizations that allow many of these tables to be “shared”). This set of virtual-function tables allows base-class functions to be overridden by functions that are as “high up” as `X` in the class hierarchy. The virtual-function table for `X` itself remaps each virtual function in `X` to itself.

When an object of class `C` is created, all these virtual-function tables are installed within `C` and its bases as appropriate. See §11.1: *Constructors* on page 165 for a more precise specification of the timing of this installation.

The virtual call mechanism is *skipped* when the `::` operator is used in conjunction with the `.` or `->` operator to access the function:

```
A[i]->shape::draw();
```

Here the `shape::` prefix means “really call the draw function in `shape`”. This would produce:

```
Nothing interesting here.
```

for each drawing in Example 9.6.

*“Sticky” virtual* If member function `f` in a derived class `X` overrides a virtual function in a base, the **virtual** specifier is implied for `X::f`. That is, **virtual** is “sticky”; once said, you cannot get rid of it among overriding functions.

*Pure declaration* A virtual function in a class used as a base must be defined somewhere, or must be declared *pure*. A pure declaration can be given for a virtual member function, and consists of supplying “= 0” in the *member\_declarator*:

```
struct shape {
    virtual void draw() = 0;
};
```

This is a useful alternative to the way `draw()` was written in Example 9.6, because it eliminates the function body that will never get called anyway:

```
virtual void draw() {
    printf("Nothing interesting here.\n"); }
```

The result is that if `draw()` is called for a `shape` object that is not part of a derived object, the value 0 (zero) is called — resulting in a memory fault on most machines. But if you design your program so that never happens, you can usefully omit the dummy function body by using the pure declaration.

---

## 9.8 Abstract Classes

A class is called *abstract* if it has at least one pure **virtual** function. No objects of an abstract class can be created except as a base of a non-abstract derived class. You cannot declare an object of an abstract class type. An important use of an abstract class is to provide an interface without providing any implementation; an implementation can be provided in a derived class.

---

## 9.9 Virtual Base Classes

**virtual functions** versus **virtual bases** A base class is *virtual* if the **virtual** key word is supplied when the base class is mentioned in the definition of the derived class. Such a base class is known as a *virtual base*. Do not confuse **virtual** functions with **virtual** bases. The meanings of the keyword are unrelated in the two contexts.

*Multiple virtual base classes* If a base class **B** occurs more than once as a direct or indirect virtual base class of **D**, all such occurrences refer to exactly one copy of the base class. Example 9.7 shows this.

**Example 9.7**


---

```

struct s1 { };
struct s2 { int f; };
struct s3:s1, virtual s2 { };
struct s4:s1, virtual s2, s3 { };
struct s5:s4, s2 { };

```

---

In Example 9.7, `s2` is mentioned as a virtual base class of `s3` and of `s4`, and as a non-virtual base class of `s5`. In class `s4` there is exactly one base `s2`, shared between `s4` and `s4`'s base class `s3`. In class `s5` there are two occurrences of `s2`: one as part of `s4` and one as part of `s5`. The declaration of `s5` is unusual in that there is no way to refer to `s2` members of `s5`, because all such references are ambiguous. However, it is possible to refer to the `s2` members of `s4` with the `s4::` scope resolution operator. For example:

```

s5 x;
x.f = 1;           // Ambiguous: Which s2::x?
x.s4::f = 1;      // OK, the single s2 in s4

```

Virtual base classes are interesting only in the context of multiple inheritance. In the above examples they allow local sharing of information among the `s3` and `s4` classes.

*Virtual base pointer* Virtual base classes complicate conversion of a derived class pointer to a base class pointer, and thus complicate calling a member function inherited from a virtual base class (which requires `this` to refer to the base class). In terms of the implementation, any class with a direct virtual base class contains a pointer (called a *virtual base pointer*) that points to that virtual base class. The conversion from the derived class pointer to the base class pointer essentially just accesses that virtual base pointer.

When an object is created, enough room is allocated for just one copy of a virtual base class. The virtual base pointer for that object and for any base classes containing the virtual base class are initialized at the creation (see §11.1: *Constructors* on page 165). Pursuing the previous examples, consider the declaration of an `s4` object:

```

s4 x;

```

The space allocated accommodates one `s2` object. `s4`'s and `s3`'s virtual base pointers for `s2` are both initialized to point to that `s2` object. Therefore, any reference to `this->f` (`f` is in `s2`) in a member function of either `s4` or `s3` refers to the same `int f`.

## 9.10 Pointers to Members

You can declare pointers to members of a class. A pointer-to-member declaration uses the following syntax from *ptr\_operator* (see Chapter 5: *Declarations and Definitions*) in its *declarator*:

```
{identifier ::}... *
```

The class containing the members to which the pointer-to-member points is identified by this syntax, which is basically a *type\_specifier* referring to a previously declared class. For example:

```
struct s {
    int i, j, k;
    float f;
};
int s::* p;
```

In this example `p` is “pointer-to-member-of-`s`-of-type-`int`”. The `s::` shows that the pointer points to members of `s`, and the leading `int` tells you that those members are of type `int`.

*When to use  
pointers to members*

We already have pointers in C, so what good is it to tie down pointers to point to specific class members? Does that make any sense?

Consider the following problem. Write a function to take an array of `s`'s and initialize the `i` member of each array element to 0 (zero):

```
void init_0(s A[], int elements) {
    for (int n = 0; n < elements; n++)
        A[n].i = 0;
}
```

This is straightforward. But suppose you want to choose whether to initialize the `i`, `j`, or `k` member. You could pass `init_0` a “choice” parameter:

```

void init_0(s A[], int elements, int which ) {
    for (int n = 0; n < elements; n++)
        switch(which) {
            case 0: A[n].i = 0; break;
            case 1: A[n].j = 0; break;
            case 2: A[n].k = 0; break;
        }
}

```

But every time you add a new member to `s`, you have to change `init_0`. An alternative is use a pointer-to-member. A pointer-to-member is essentially a way to represent the name of a class member.

```

void init_0(s A[], int elements, int s::*p) {
    for (int n = 0; n < elements; n++)
        A[n].*p = 0;
}

s A[10];
main () {
    init_0(A, 10, &s::i); // Initialize the i field.
    init_0(A, 10, &s::k); // Initialize the k field.
}

```

*Offset of a member* With respect to implementation, a pointer-to-(data-)member is essentially just the offset from the beginning of any class object to that member; it is really just a way to represent the ANSI Standard C `offsetof` macro in a strongly typed manner. When `f` uses the `.*` operator on `A[n]` and `p`, it takes the base address of `A[n]`, adds the offset contained within `p`, and arrives at the address of the particular member of `s`; that member is then assigned the value `0` (zero).

Because of the strong typing, you cannot pass `init_0 &s::f`, because `&s::f` is of type pointer-to-member-of-`s`-of-type-`float`, and `init_0` takes pointer-to-member-of-`s`-of-type-`int`.

You can also have a pointer-to-member of a function type. This allows you to take the address of a non-static class member function; this is different from the standard C pointer-to-function, because the latter cannot point to non-static class member functions.

For example:

```
struct s {
    int fnc1(), fnc2();
};
int (s::* p)() = &s::fnc1;
```

Here `p` is a pointer-to-member-of-`s`-of-type-function-returning `int`. We initialized it to `&s::fnc1`. We can later supply an object of type `s` and call the pointed-to function:

```
main () {
    s x;
    (x.*p)();    // Same as x.fnc1()
    p = &s::fnc2;
    (x.*p)();    // Same as x.fnc2()
}
```

*Pointers to static member functions*

Pointer-to-member functions are not appropriate for taking the address of static member functions. Static member functions do not need the “hidden” argument of the class type that is passed to non-static member functions. Thus straightforward C pointers-to-functions can point to static member functions:

```
struct s {
    int fnc1(), fnc2();
    static int fnc3();
};
int (* p)() = &s::fnc1;    // Not valid!
int (* q)() = &s::fnc3;    // OK
```

# Access Control in C++

---

*Access control* determines who can use a name declared in a class or inherited from another class. For example, a class can implement a data type and allow only functions that manipulate objects of the class type to be visible; the data implementing the type can be hidden by denying anyone other than the class implementor access to that data. Access control allows specification of access to individual names in a class, and can discriminate which among all other classes can access the name.

---

## 10.1 Access

Each member of a class has one of the following accesses, in diminishing order of “usability”:

<b>public</b>	A <b>public</b> name is accessible in any context.
<b>protected</b>	A <b>protected</b> name owned by a class <b>X</b> can be used by a member function defined for <b>X</b> or any class immediately derived from <b>X</b> , and by any friend of <b>X</b> (see §10.4: <i>Friends</i> on page 159).
<b>private</b>	A <b>private</b> name owned by class <b>X</b> is accessible by a member function defined for <b>X</b> and by any friend of <b>X</b> .
inaccessible	An inaccessible name is not accessible in any context.

**Example 10.1**

---

```

struct B {
public:
    int i_pub;      // i_pub is public.
    void bfunc();  // bfunc is public, too.
protected:
    int i_prot;    // i_prot is protected.
private:
    int i_priv;    // i_priv is private.
};

void B::bfunc() {
    i_priv++;      // OK: Member function can access
                  // private members.

    i_prot++;      // OK: Member function can access
                  // protected members.
}

void anyfunc() {
    B b;
    b.i_priv++;    // Error; cannot access
                  // private member
    b.i_prot++;    // Error; cannot access
                  // protected member
    b.i_pub++;     // OK: Anyone can access
                  // a public member.
    b.bfunc();     // OK, for same reason;
                  // increments i_priv
}

```

---

In Example 10.1 `bfunc()` is provided as the only way an “outsider” can increment the private member `i_priv`. In this manner the implementor of class `B` can completely control both the reading and the writing of its data members. We will consider the protected member `i_prot` on page 156.

*Specifying the  
access of a member*

The access of a member is specified through a combination of the following:

- an *access\_specifier* within a *class\_definition*; that, is one of the following:



```
private:
public:
protected:
```

- the access specifier for a base class: you can specify **public**, **private**, or **protected** for each base class in the list of base classes for a derived class (a base is a **public**, **private**, or **protected** base of another class)
- through an access override (see §10.2: *Access Override* on page 157)

#### *Using access specifiers*

The *access\_specifier* specifies the access control for declarations physically following it in the class declaration (unless the declaration is instead interpreted as an access override), up until the next *access\_specifier*. For **struct** or **union**, **public:** is implied at the beginning of the **struct** or **union** declaration. For **class**, **private:** is implied.

#### *Determining access of inherited members*

When a class derives from a base class, the members of the base class are inherited as members of the derived class. The access of those members in the derived class is determined from the base access specifier and the access of the member in the base class. You can view the base access specifier as a filter that transforms the access of a name in the base class to a new access as the name becomes a member of the derived class. In the absence of a base access specifier, **public** is implied if the derived class is a **struct** or **union**, and **private** is implied otherwise. Here is how to determine the access of inherited members in a class **D** derived from a base **B**:

- If **B** is a **public** base of **D**, any **public** or **protected** member of **B** is a **public** or **protected** member, respectively, of **D**; that is, the access is not changed — and any **private** or inaccessible member of **B** becomes an inaccessible member of **D**.
- If **B** is a **protected** base of **D**, any **public** member of **B** is “demoted” to a **protected** member of **D**, the access of a **protected** name is unchanged, and any **private** or inaccessible member of **B** becomes an inaccessible member of **D**.
- If **B** is a **private** base of **D**, any **public** or **protected** member of **B** is “demoted” to a **private** member of **D**, and any **private** or inaccessible member of **B** becomes an inaccessible member of **D**.

*Summary of the derivation rules* Any derivation makes **private** and inaccessible members inaccessible, and demotes any other access to a level no more “accessible” than the level of the derivation. See Table 10.1.

**Table 10.1** *Access of Members of Base and Derived Classes*

Access of member in base class	Access of member in derived class when base access is:		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private or inaccessible	inaccessible	inaccessible	inaccessible

Consider the three possible derivations — **public**, **protected**, and **private** — from class **B** in Example 10.1 on page 152. Here are the accesses for the inherited members, by the derivation rules previously described and illustrated:

```
In B:  i_priv, f_priv: private
       i_prot, f_prot: protected
       i_pub , f_pub : public

struct D : private B { };
```

```
In D:  i_priv, f_priv: inaccessible
       i_prot, f_prot: private
       i_pub , f_pub : private

struct D : protected B { };
```

```
In D:  i_priv, f_priv: inaccessible
       i_prot, f_prot: protected
       i_pub , f_pub : protected

struct D : public B { };
```

```
In D:  i_priv, f_priv: inaccessible
       i_prot, f_prot: protected
       i_pub , f_pub : public
```

To determine whether an attempt to use a class member is valid, this is all you need to know:

- the class to which the member belongs
- the access of the member (**public**, **protected**, **private**, or inaccessible)
- the accessor: who is accessing the member

*To which class does the member belong?*

In any use of a class member, you must take care to identify the class to which the member belongs. Because of C++'s inheritance, the same name can be used for a member of the class in which it is defined, and also for a member of some derived class that inherited it. When you write:

```
T::x
```

**x** is a member of **T**, even though it might have been inherited from some base class. For:

```
b.x
```

**x** is a member of **b**'s type. For:

```
p->x
```

the type of **\*p** is the owner. Finally, for just:

```
x
```

there is usually an implied **this->**, as in a member function. **x** then is a member of the type of **\*this**.

In any of these cases, **x** might be defined originally in some base class, but it is being referenced as a member of a derived class.

Consider class **D** with **B** as a private base class:

```
struct D: private B {
    void dfunc();
};
// Recall that as members of D,
// i_priv is inaccessible,
// i_prot is private, and
// i_pub and bfunc are private.
void D::dfunc() {
    i_priv++;           // Error: this->i_priv is
                        // an inaccessible member of D.
```

```

    i_prot++;           // OK:  this->i_prot is
                        //   a private member of D.
    i_pub               // OK:  this->i_pub is
                        //   a private member of D.

    B b;
    b.i_priv++;         // Error: cannot access
                        //   private member of B
    b.i_prot++;         // OK:  A derived class can access
                        //   a protected member of its base
    b.i_pub++;          // OK
    int B::*p = &B::i_prot; // OK: can access
                        //   protected member of base
}

```

Here `D::dfunc` is accessing `i_prot` once as a member of `D` and twice as a member of `B`. For the unadorned reference to `i_prot`, `this->` is implied, and so `i_prot` is considered a member of `D` (the type of `*this`). Because `i_prot` is a private member of `D`, and `dfunc()` is a member function of `D`, the access is allowed. For the references `b.i_prot` and `B::i_prot`, `i_prot` is a member of `B`. The access is allowed for a different reason: the protected members of a base are accessible by an immediately derived class (`D`).

#### *Name overloading and access*

In C++ a single function name can have different meanings. This is called *overloading* (see Chapter 12: *Name Overloading and Operator Functions in C++* for more information). Because names can be overloaded, access is determined for a particular member of a class, not just by the member's name.

```

struct G {
public:
    int f(int);
protected:
    int f(double);
private:
    int f(char *);
};
G g;

```

In the preceding example:

- `g.f(1)` is an access to a **public** name owned by `G`.
- `g.f(1.0)` is an access to a **protected** name owned by `G`.
- `g.f("1")` is an access to a **private** name owned by `G`.

Access restrictions are computed based on a member of a class, not just a name, because the name can be overloaded.

## 10.2 Access Override

Access override allows preserving (that is, leaving unchanged) the access of a **public** or **protected** name inherited from a base class, despite the access specifier attached to the base-class name in the class definition. Any member declaration with no *decl\_specifiers* and for which the declaration is of the form *identifier::simple\_name* is an *access override* and does not actually declare any member of the class. The *identifier::* in the access override must denote one of the base classes.

Again, taking class **B** in Example 10.1 on page 152 and a sample derivation **D** from **B**, we have:

```
struct D : protected B { };
```

The names in **D** have the following access rights:

```
In D:   i_priv: inaccessible
        i_prot: protected
        i_pub , bfunc: protected
```

You might want the public members of **B** to remain public. You can do so by using access override:

```
struct D : protected B {
    public:  B::i_pub;  B::bfunc;
}
```

The names in **D** now have the following access rights:

```
In D:   i_priv: inaccessible
        i_prot: protected
        i_pub , bfunc: public
```

Deriving **E** from **D**, we keep the protected member of **D** protected despite a **private** derivation:

```
struct E : private D {
    protected: D::i_prot;
};
```

The names in **E** have the following access rights:

```
In E:   i_priv: inaccessible
        i_prot: protected
        i_pub, bfunc: private
```

Otherwise, **i\_prot** would have been **private**.

*Access override for  
overloaded  
functions*

An access override for a name adjusts the access to all declarations of that name in the base class. For an overloaded function, the access is adjusted for all functions in the base class. Note that all functions must have the same access in the base class:

```
struct H {
    public:    int f(int), f(double);
    protected: int g(int);
    public:    int g(double);
};

struct DH: private H {
    public:    H::f; // OK, preserve public access
    protected: H::g; // Not valid! Different
                  // accesses for g
};
```

You can only *preserve* the same access, not *change* it. Here is an example of an invalid attempt:

```
struct DH: private H {
    protected: H::f // Not valid; cannot change
                  // public to protected
};
```

---

## 10.3 Multiple Access Through Virtual Base Classes

Due to sharing between virtual base classes, a name can be accessible through more than one path. The access granted is the one that allows the greatest accessibility.

```
struct V: { int x; };
struct D1: virtual public V { };
struct D2: virtual private V { };
struct E: D1, D2 { }
```

Here, `E::x` has public access in `E` because there is a path through `D1` to obtain `v` publicly. The path through `D2` is ignored.

---

## 10.4 Friends

A *friend* is a function that can access members of a class that are not **public**, even though the function is not a member of the class.

### Example 10.2

---

```
class position {
    int where[2];
};
class attack;
class attack_pieces {
    friend attack; // Declares attack to be a friend
    int queens_rook[2];
    int queens_knight[2];
    int queens_bishop[2];
    int queen[2];
    int kings_bishop[2];
    int kings_knight[2];
    int kings_rook[2];
    int in_check(int *);
public:
    attack_pieces();
    analyze_knights(int,int);
};
class attack {
public:
    void move_knight(int *, int *);
    void move_bishop(int *, int *);
    void move_rook (int *, int *);
```

```

    void move_queen (int *, int *);
};
class move_piece : attack {
    int legal_moves[100];
    int check_legal( int where[]);
}

```

---

In Example 10.2, every function in **class** `attack` is a friend of **class** `attack_pieces`. The functions of `attack` can access and change all the private members of `attack_pieces`. The declaration **friend** `attack` declares the **class** `attack` as a friend of `attack_pieces`. The meaning of a class as a friend is that all the class functions are friends of the class declaring the **friend** class. The name `move_knight` is not in the scope of `attack_pieces`, nor are any of the other member functions of `attack`.

```

main(){
    position whereami;
    attack_pieces mine, yours;
    attack mymove, yourmove;
    // Not valid:
    mine.move_knight((int *)position.where);
}

```

Because `move_knight` is not in the scope of `attack_pieces`, you cannot reference it as though it is a member of `attack_pieces`.

The function `check_legal()` is not a friend of `attack_pieces`, even though it is derived from **class** `attack`.

The property of being a friend is not necessarily reflexive, nor is “friendship” transitive. In the example, `attack` is a friend of `attack_pieces`, but `attack_pieces` is not a friend of `attack`.

You can make an individual member function of `attack` a friend of `attack_pieces`, rather than all functions in `attack`, as illustrated in Example 10.3.



**Example 10.3**

---

```

class position {
    int where[2];
};

class attack {
public:
    void move_knight(int *, int *); // This is the
                                   // only friend.

    void move_bishop(int *, int *);
    void move_rook   (int *, int *);
    void move_queen  (int *, int *);
};

class attack_pieces {
    friend void attack::move_knight(int *, int *);
    int queens_rook[2];
    int queens_knight[2];
    int queens_bishop[2];
    int queen[2];
    int kings_bishop[2];
    int kings_knight[2];
    int kings_rook[2];
    int in_check(int *);
public:
    attack_pieces();
    analyze_knights(int,int);
};

```

---

In Example 10.3, only the function `move_knight()` is a friend of `attack_pieces`. The other member functions of `attack` have no access to **private** or **protected** members of `attack_pieces`. If a **friend** declaration references an overloaded name or operator, only the function that is referenced by the argument types becomes a **friend**.

Any **friend** function whose body is defined in a **class** declaration is automatically **inline**.



# Special C++ Member Functions

---

You create types via classes, and classes have special member functions designed to protect the integrity of such types. For example, class constructors (see §11.1: *Constructors* on page 165) are special member functions that operate in conjunction with other C++ language rules to ensure that a class object is not used until it is properly initialized. In C we might specify a stack object via the functions:

```
// Implement stacks with integer "handles":
typedef int stack;
stack create_stack();
void push(stack S, int value);
int pop(stack S);
```

You might use this stack object as in the following function:

```
void f() {
    stack S;
    S = create_stack();
    push(S, 1);
    if (pop(S) != 1) error("What?!");
}
```

But if we forget the call to `create_stack()`, `S` is uninitialized, which will cause calls to `push()` and `pop()` to fail. You can define a constructor member function for stacks. Such a constructor is always called when a stack object is declared. This can ensure that a stack object is always initialized before it is used.

Table 11.1: *C++ Special Members* lists the kinds of special members in C++ and indicates their purpose.

**Table 11.1 C++ Special Members**

Special Member	Purpose
Constructor	Properly initialize an object
Destructor	Properly “destroy” an object when the object ceases to exist; destruction can include, for example, memory deallocation
Copy constructor	Control what happens when a copy of an object is made; a copy constructor is a special kind of constructor
<code>operator =()</code>	Control what happens when something is assigned into an object
Conversion function	Define automatic conversions from a class object to another type, the same way C++ automatically converts from (for example) <code>double</code> to <code>int</code> when necessary
<code>operator new</code>	Control heap allocation of an object
<code>operator delete</code>	Control heap deallocation of an object

All special members are invoked automatically without specifically referring to them. For example:

```

void f() {
    stack S1, S2; // Invokes constructor twice
    S1 = S2;      // Invokes stack::operator =
    stack S3 = S1; // Invokes copy constructor
    stack *p = new stack;
                  // Invokes stack::operator new
                  // and the constructor
    delete p;     // Invokes stack::operator
                  // delete
    int i = S1;   // Invokes stack::operator int
}

```

The member is invoked only if it is declared for the `stack` class. Notice that in the example, no specific class member is named; each member is invoked automatically by the compiler. In that sense these members are “special”.

*Access rules for special members*

Special member functions obey all the usual access rules. For example, if a `stack` constructor is private, `stacks` cannot be declared within functions that are not members of the `stack` class. If you try to declare a `stack` in a function

that is not a member of the stack class, you get an access error when the private constructor is called.

---

## 11.1 Constructors

*Constructors create class objects*

A *constructor* is a member function with the same name as its class. A class can have any number of constructors, or even none at all. A constructor's declaration cannot specify a return type.

*A suitable constructor is chosen by overload resolution*

If a class type has a constructor, creation of an object of that class type invokes a suitable constructor. The term *suitable constructor* means that one constructor from among those declared will be called. Overload resolution (see §12.1: *Overload Resolution* on page 191) is used to choose such a constructor. The arguments to the constructors are taken from the context in which the object is being created.

Creation of a class object by a constructor can occur in the following ways:

- declaration of a class object as a local or global variable
- explicit use of **operator new**
- explicit call of a constructor
- creation of a temporary class object by the compiler
- as a member of a class that is being created
- as a base of a class that is being created

*Constructors ensure that objects are initialized*

This ensures that an object is initialized before any use is made of the object. Essentially, the constructor takes “raw memory” and turns it into a valid class object. The special behavior of constructors is different from anything you can easily mimic. The rules are carefully designed to ensure that no member function is called for a class object until that class object is fully initialized.

*Characteristics of constructors*

A constructor must be a non-**static** member. The **this** parameter passed to the constructor is the pointer to the area of storage that the constructor turns into a valid object. A constructor for a given class differs from all other non-**static** member functions of the class, because the **this** parameter passed to a constructor points to storage that has not been turned into a valid object, whereas for all other member functions, **this** points to a valid class object.

A constructor cannot be declared **const** or **volatile**. A constructor cannot be **virtual**. You cannot take the address of a constructor.

*Default constructor* If a class has no default constructor, but one is needed (because work such as initializing virtual-function tables is necessary for constructing an object), the compiler generates such a default constructor, with **public** access.

*Constructor arguments* You can provide arguments to the constructor in the following contexts:

- in a declaration of an object with a parenthesized expression list
- in a **new** expression with optional parenthesized expression list
- in a constructor initializer (see §11.1.1: *Constructor Initializers* on page 170) with a parenthesized expression list

An explicit invocation of a constructor includes zero or more argument expressions, just as in the invocation of any member function.

No other contexts exist for which arguments to a constructor are present. For any cases of object creation that demand a constructor where it is not possible to specify any arguments, zero arguments are implied. This means that overload resolution will be forced to find a constructor that is callable with no arguments. Such a constructor is called a *default constructor*. For example:

```

struct stack {
    stack();           // Default constructor, takes no
                      // arguments
    stack(int);        // Not a default constructor
    stack(int,int);    // Also not a default constructor
};

void f1() {
    stack S1;          // Calls default constructor
                      // stack()
    stack S2(3);       // Calls stack(int)
    stack S3(3,3);     // Calls stack(int,int)
    stack S4("xyz");   // Error: Overload resolution
                      // cannot find a suitable
                      // constructor.
    stack S5(4.5);     // Calls stack(int) after
                      // converting 4.5 to an int
                      // via standard conversions

```

```

stack *p1 = new stack;           // Calls stack()
stack *p2 = new stack(10);       // Calls stack(int)
stack *p3 = new stack(1,2);      // Calls stack(int,int)
somefunc(stack());              // Explicit call to stack()
somefunc(stack(1));              // Explicit call to stack(int)
somefunc(stack(1,2))             // Explicit call to
                                // stack(int,int)
}

```

*What  
constructors do*

When a class contains members or bases that are classes, the construction of an object of that class type requires that the members or bases are constructed as well. In addition to your code for a constructor, the High C/C++ compiler inserts other code that causes the class type's members and bases to be appropriately constructed. All such inserted code occurs ahead of your source code in a constructor.

If we ignore the issue of virtual bases for the time being, this is what a constructor for a class does:

1. Calls a suitable constructor for each direct base of the class in left-to-right order.
2. Calls a suitable constructor for each member of the class.
3. Installs virtual-function tables for this class and its bases (see the **High C/C++ Programmer's Guide.**)
4. Executes your constructor code.

The reason the bases and members of the class are constructed before executing your constructor code is that your code might refer to a member or a base's member. These members must be properly initialized for your code to work correctly. C++ thus guarantees that the member or base objects are properly initialized before any attempt to access them in your constructor code.

*Constructing a  
direct base*

Construction of a direct base invokes a constructor defined in the base's class. In turn, the constructor in the base class, following the rules above, constructs its own bases and members. In this manner, all members, bases, and members of bases are constructed before your constructor code is executed.

*Constructors and  
virtual functions*

If a constructor for class **X** calls a **virtual** member function of **X**, the call is always to the actual function in **X**, not to a function in a class derived from **X**. It would be improper for a constructor to call an overriding member function

in a class derived from **X**. This is because, due to the order of construction, that derived class is not fully constructed yet, and so the behavior of any overriding function might be unpredictable.

Example 11.1 shows how a **virtual** function is used in constructor code.

### Example 11.1

---

```

struct s1 {
    s1() {
        f1();           // Calls s1::f1 always
    }
    virtual void f1() { x = 1; }
    void init_s1() {
        f1();           // May call overriding function
    }
    int x;
};

struct s2 {
    s2() {
        f2();           // Calls s2::f2 always
    }
    void f2() { x = 2; }
    void init_s2() {
        f2();
    }
    int x;
};

struct s3: s1 {
    s3() {
        f3();           // Calls s3::f3 always
        f1();           // Calls s3::f1, not s1::f1
    }
    void f1() { s1::x = 30; } // Overrides s1::f1
    void f3() { x = 3; }
    void init_s3() {
        init_s1();       // Calls init_s1 for base s1
        y.init_s2();     // Calls init_s2 for member y
        f3();
        f1();
    }
};

```



```

    int x;
    s2 y;
};
void main() {
    s3 obj;
    obj.init_s3();
}

```

---

When `obj` is declared to be **class** type `s3`, in Example 11.1 on page 168, the generated code does the following:

1. Calls constructor `s1::s1()` for base `s1`, which installs virtual tables for `s1` that remap `s1`'s functions to themselves. Calls `f1()`. This means call `s1::f1()`.
2. Calls constructor `s2::s2()` for member `y`: which installs virtual tables for `s2` that remap `s2`'s functions to themselves. Calls `f2()`. This means call `s2::f2()`.
3. Installs virtual tables for `s3` that remap `s3`'s functions to themselves, and remap `s1::f1` to overriding `s3::f1`.
4. Calls `f3()`. This means call `s3::f3()`. Calls `f1()`. This means call `s3::f1()`, not `s1::f1`, because the latter has been overridden.

So, the non-constructor functions are called in this order:

```

s1::f1()
s2::f2()
s3::f3()
s3::f1()

```

Installing virtual-function tables in step 3 reinstalls virtual-function tables for base `s1`. In general, base-class virtual-function tables are overwritten during construction as many times as the length of the derivation chain from the base to the topmost object being declared.

By contrast, suppose you write your own mechanism for initialization by defining the `init_xx()` functions in the classes in Example 11.1 on page 168. The call in `main()` to `obj.init_s3()` behaves differently from the construction of `obj`, even though you try to write the `init_xx()` functions to mimic the constructors. The reason is the timing of virtual-function table installation. By the time your program calls `init_s3()`, the virtual-function

tables installed are those for `s3`, so that `s1`'s functions have been overridden. This causes the generated code to do the following when calling `obj.init_s3()`:

1. Calls `init_s1()` for base `s1`, which calls `s3::f1()`, which overrides `s1::f1`.
2. Calls `init_s2()` for member `y`, which calls `s2::f2()`.
3. Calls `f3()`. This action calls `s3::f3()`.
4. Calls `f1()`. This action calls `s3::f1()`.

So, the non-constructor functions are called in this order:

```
s3::f1()
s2::f2()
s3::f3()
s3::f1()
```

The first function called is an overriding one, whereas in the case for constructors, the first function called is `s1::f1()`.

---

### 11.1.1 Constructor Initializers

*Supplying arguments to a constructor* A constructor for a class automatically constructs its direct bases and its members. Overload resolution always chooses default constructors, because no arguments are given to the base and member constructors. However, a *constructor initializer* is a mechanism you use to specifically supply arguments to the constructors, so that overload resolution can choose something other than a default constructor.

After a constructor's declarator, and preceding its definition (`{...}`), you can supply a colon and a list of one or more initializers separated by commas. Each initializer can take the form of an identifier followed by a parenthesized expression list. The identifier must refer to a member, a direct base, or a virtual base. The expressions in the list are taken as the arguments for the constructor to be used for the base or member referenced by the identifier. If the member referenced is not a class object, the list must have exactly one expression, and this expression is taken as the initial value for the member.

*Give bases and  
members different  
names*

If a member and a base have the same name, the member is preferred in the constructor initializer, and there is no way to specify the base. Giving a base and a member the same name is not a recommended practice.

The constructor initializer does not change the order of initialization of bases and members for constructors. It just allows other than the default constructor to be chosen for the initialization of a base or member.

Consider adding the following two constructors to Example 11.1 on page 168:

```
void s1::s1(int i) { x = i; }
void s2::s2(char *s) {
    printf("whaddaya know:%s\n",s);
}
```

This addition allows you to modify the definition of the constructor in `s3` to call each of these two non-default constructors rather than the default constructors called in the following example:

```
struct s3: s1 {
    s3() : s1(5), y("called from s3") {
        ...
    }
}
```

So, when an `s3` object is constructed, `s1::s1(int)` is called for the base `s1`, and `s2::s2(char *)` is called for the `y` member — rather than calling `s1::s1()` and `s2::s2()`, as is done without the constructor initializer.

A constructor initializer is also useful for initializing non-`static const` members and reference members of a class, as in the following example:

```
int i;
struct s {
    s() : x(i), c(3), k(7.2) { }
    int &i;
    const int c;
    float k;
};
```

This initialization of field `k` could have been moved into the body of `s()` with an assignment statement (`k = 7.2`), but this is not possible for the `const` or reference member.

---

### 11.1.2 Virtual Base Classes and Constructors

Virtual bases are a special case for constructors. There is only one copy of any virtual base, so that base is initialized by the “topmost” constructor being called. Constructors called by the topmost constructor for initializing bases are “instructed” that they must not construct any virtual base; this assures that only the topmost constructor constructs the virtual base. After all, the topmost object being constructed is the one that must allocate the storage for the virtual base anyway.

Constructors that are not topmost are “instructed” not to initialize a virtual base by being passed a hidden parameter that tells them whether or not they are a topmost constructor. Given this action, the procedure for a constructor for a class **X** can be rewritten (in pseudo-code) as follows:

---

**Pseudo-code: Constructor for Class **X****

---

```
constructor_code(boolean I_am_topmost) {  
    0. if (I_am_topmost) {  
        Initialize all virtual base pointers for all bases. For each base in  
        a depth-first, left-to-right traversal of the class graph, construct  
        each virtual base and pass FALSE as the value for  
        I_am_topmost.  
    }  
    1. Call a suitable constructor for each direct, nonvirtual base of X in  
    left-to-right order. Pass FALSE as the value for I_am_topmost.  
    2. Call a suitable constructor for each member of X. Pass TRUE as the  
    value for I_am_topmost.  
    3. Install the set of virtual-function tables for class X (this includes vir-  
    tual-function tables for X’s bases; see §9.7: Virtual Functions on  
    page 141).  
    4. Execute your constructor code.  
}
```

---

*A topmost constructor initializes all virtual bases* A constructor initializer can specify the initialization for a virtual base because a topmost constructor initializes all virtual bases. This implies that any constructor initializer for a virtual base will be ignored if that initializer is part of a constructor that is not topmost. For example:

```

struct V {
    V(char c) { ch = c; }
    char ch;
};

struct A: virtual V {
    A(): V('a') { }
};

struct B: virtual V {
    B(): V('b') { }
};

struct X: A, B, virtual V {
    X() : V('c') { }
    A a;
};

main () {
    X x; // Uses V('c') for base V, ignores V('a')
        // in A() and V('b') and B()
        // Uses V('a') for member a
    printf("%c %c\n", x.ch, x.a.ch); // Prints c and a
}

```

Expanding what happens when `x` is constructed reveals the following steps (refer to the `constructor_code` procedure above):

### **Pseudo-code: Constructor for Members of `X`**

---

0. Since `I_am_topmost` is `TRUE`, initialize the virtual base pointers within `A`, `B`, and `X` to point to the location of `V`.  
Construct `V` passing argument `'c'`.
  4. Execute `V::V()`'s user-written code (`"ch = c"`).
1. Construct `A` with `A::A()`:
  0. Since `I_am_topmost` is `FALSE`, do nothing.
  4. Execute `A::A()`'s user-written code.

Construct **B** with **B::B()**:

0. Since **I\_am\_topmost** is **FALSE**, do nothing.
  4. Execute **B::B()**'s user-written code.
  2. Construct member **a** with **A::A()**:
    0. Since **I\_am\_topmost** is **TRUE**, initialize the virtual base pointers within **A** and **V** (as constituents of member **a**), to point to the location of the **V** within **a**.  
Construct **V** within **a**, passing argument '**a**'.
      4. Execute **V::V()**'s user-written code ("**ch = c**").
    4. Execute **A::A()**'s user-written code.
  4. Execute **X::X()**'s user-written code.
- 

The **printf** statement at the end prints out the characters **c** and **a**.

---

### 11.1.3 Copy Constructors

Some constructors, called *copy constructors*, are used to control the semantics of copying a class object in so-called “initialization” contexts. A *copy constructor* for a class is any constructor that can be called with a single argument of the same class type. The first argument of a copy constructor must be a reference type, as in the following example:

```
// Valid copy constructors for class X:
X::X(const X&);
X::X(X&, float f = 1.2)
// An erroneous copy constructor:
X::X(volatile X)
```

*When objects are copied* The initialization contexts are contexts in which an object is copied, except for assignment. Specifically, these are the contexts:

- forms of user-specified initialization in which `=` is used
- function argument passing
- function value return

### Example 11.2

---

```

struct s {
    s(s&);
    s(int);
};

s f(s);      // f takes and returns an s.
s x;
s y = x;     // Calls copy constructor to copy x into y
              // (that is, y takes the value s(x))
f(y);        // Calls copy constructor to copy y to f's
              // argument (that is, the argument takes
              // the value s(y))
s f(s parm) { return parm; }
              // Calls copy constructor to copy parm to
              // return result (that is, s(parm) is
              // returned)
s z(1);      // Does NOT call copy constructor;
              // calls s(1) to construct z

```

---

*Compiler-generated copy constructors* The compiler generates a **public** copy constructor if you do not declare one in a class. If all bases and members of a class have copy constructors accepting **const** arguments, the generated copy constructor takes a single argument of type **const class\_name&**. Otherwise, it takes a single argument of type **class\_name&**, and therefore copying a **const class\_name** object is not possible. The semantics of a generated copy constructor are to copy each member and base via the appropriate copy constructor for the member or base. In practice, the High C++ compiler generates fast bitwise copy semantics for copying class objects when you have not defined any copy constructors.

*Initializing declarations* When you specify an initialization that uses the parenthesized list of initialization expressions, copy constructors are not used, as exemplified by the

declaration of `z` in Example 11.2 on page 175. It is helpful to contrast several different initializing declarations:

```
s z1(1);
s z2 = s(1);
s z3 = 1;
```

The first declaration calls the `s(int)` constructor directly to form `z1`. The second one calls the `s(int)` constructor to form `s(1)`, and then calls the copy constructor on the result to form `z2`. The third is the same as the second, except that the compiler provides the conversion of `1` to `s` via the constructor. See also §5.7: *Initialization* on page 72.

*Bitwise copy  
semantics allow  
eliminating  
constructor calls*

Using `=` directs the program to construct the value specified in a temporary object, and then copy the temporary to the variable being initialized via a copy constructor. If the compiler generates the copy constructor itself, and if it notices the semantics of the copy constructor are bitwise copy, the compiler can eliminate the call to the copy constructor and make `s z2 = s(1)` as efficient as `s z1(1)`, directly calling `s(int)` to construct `z2`. But where the semantics of the copy constructor are not bitwise copy, the optimization is not possible.

---

## 11.2 Destructors

A *destructor* for a class does the reverse of what a constructor does. It takes a valid class object and does anything necessary to destroy the object. For example, it might free up heap storage that was allocated during construction. A destructor for a class is a member function whose name is the same as the class name, prefixed with the character `~`. For example:

```
class buf {
    void *storage;
public:
    ~buf();      // This is a destructor declaration.
    buf(int);    // This is a constructor.
};

buf::buf(int howmuch){
    storage = new char[howmuch];
    *i = 3;
}
```



```

buf::~buf(){    // This is a destructor definition.
    delete [] storage;
}
void func() {
    buf B(1024);
    ...
}    // Storage is reclaimed by the destructor.

```

Here the destructor reclaims the storage allocated by the constructor.

*Destructors work in the reverse order of constructors*

Just as a constructor constructs bases and members and then executes user-written code, a destructor must execute user-written code and destroy bases and members. The order in which a destructor does these things is exactly the reverse of the order in which a constructor does them. This guarantees that the order in which objects are destroyed is the reverse of the order in which they are constructed. A destructor must free the storage associated with an object that was allocated via operator **new**.

A destructor can be **virtual** (in contrast to a constructor).

*Destructors take no arguments*

A destructor declaration cannot specify a return type or any parameters. However, no overload resolution is ever needed for a destructor, because it takes no arguments. Thus there can be at most one destructor for a class.

*When destructors are called*

Destructors can be invoked as follows:

- for a local variable that has gone out of scope
- for a global variable when program termination occurs
- through use of operator **delete**
- when a temporary object created by the compiler is no longer needed
- for a member of a class that is being destroyed
- for a base of a class that is being destroyed

This list parallels the list of cases when a constructor can be called. Here is pseudo-code for any destructor, along with the two hidden parameters that identify whether the destructor call is a “topmost” call, and whether the context of the call is the operator **delete**:

**Pseudo-code: Destructor for Class *X***

---

```

destructor_code (boolean I_am_topmost,
                 boolean I_should_delete) {

```

0. Install the set of virtual-function tables for class *X* (this includes virtual-function tables for *X*'s bases; see §9.7: *Virtual Functions* on page 141).
1. Execute the user-written destructor code.
2. Call the destructor for each member of *X*. Pass **TRUE** as the value for *I\_am\_topmost* and **FALSE** for *I\_should\_delete*.
3. Call the destructor for each direct, non-virtual base of *X* in left-to-right order. Pass **FALSE** as the value for *I\_am\_topmost* and for *I\_should\_delete*.
4. **if** (*I\_am\_topmost*) {  
     For each base in a depth-first, right-to-left traversal of the class graph, construct each virtual base and pass **FALSE** as the value for *I\_am\_topmost* and for *I\_should\_delete*.  
   }
5. **if** (*I\_should\_delete*) free the storage associated with the object by calling:  
     *X* :: **operator delete**  
     or, if there is no such operator, calling the global operator **delete**.  
   }

*Compiler-generated destructors*

If you do not specify a destructor for a class and one is needed, the compiler generates a **public** destructor. The generated destructor has, of course, none of your code.

*Destructors and virtual member functions*

Just as with constructors, the virtual-function tables for a class are set up before your destructor code is called. This means that any virtual member functions of a class that your code calls always call the actual function in a class. This subtlety about a destructor is important to remember, because a destructor cannot call an overriding member function in a derived class. This

is because, due to the order of destruction, that derived class might be partially destroyed, and so the behavior of any overriding function might be unpredictable.

*Characteristics of  
destructors*

Other points to remember:

- A destructor cannot be declared **const** or **volatile**.
- A destructor must be a non-**static** member.
- You cannot take the address of a destructor.

---

## 11.3 User-Defined Conversions

Type conversions of objects to and from class objects can be specified by constructors and by conversion functions.

Such conversions, also called *user-defined conversions*, are used in addition to C++'s standard conversions. User-defined conversions can be applied to the following expressions:

- an argument to a function
- an initializer value
- a function return value
- an explicit type conversion (cast)
- an operand of an expression
- the *expression* in:
  - **if**(expression)
  - **switch**(expression)
  - **while**(expression)
  - **for** (S expression; e2)

User-defined conversions are applied only if necessary. Their use obeys the usual access-control rules. At most one user-defined conversion is implicitly applied to a single value.

*Conversion by  
constructor*

There are two kinds of user-defined conversions. The first is conversion by constructor. A constructor for a class **X** accepting a single argument specifies a conversion from that argument type to the **X** type.

For example:

```

struct s {
    s(int);
    s(char *, float = 1.2);
};
s f(s&, s) {
    return 10;      // Converts 10 to s via s(int)
}
s x;
s y = f(3,          // Converts 3 to s as 1st arg to f
        "string"    // Converts "string" to s as second
        );          // argument to f
s y = 1;            // Converts 1 to s via s(int)
s z[3] = {10, y, 30}; // Calls s(int) for 10 and 30,
                      // and compiler-generated copy
                      // constructor for y.

```

*Conversion functions* The second kind of user-defined conversion is by a conversion function. A *conversion function* is a function with a name of the form **operator** followed by a sequence of one or more *type\_specifiers* (see §5.4: *Data Types* on page 52), followed by zero or more *\*s*. Neither a return type nor argument types can be specified for a conversion function. A conversion function must be a non-**static** member of a class. The conversion specified is from the class of which the function is a member to the type given by the name of the conversion function. In contrast to constructors, conversion functions can convert to a non-class type, and can specify a conversion from class **Cfrom** to class **Cto** without having to include in **Cfrom**'s definition a conversion constructor. For example:

```

struct s2 { };
void takes_s2(s2&);
struct s {
    operator char *();      // Converts s to char *
    operator const s2();    // Converts s to const s2
    s(int);                 // Can convert an int to an s
};
char * f() {
    s x;
    if (x) ...;             // Converts x to char *
                           // before testing
    while (x) ...;          // Ditto
}

```

```

s2 z = x;           // Converts x to s2, then
                    // uses bitwise copy
                    // (compiler-generated
                    // default) to
                    // copy result into z
printf("%c",x[3]);  // Converts to char *
                    // before subscripting
takes_s2(1);        // ERROR
return x;           // Converts to char * before
                    // returning
}

```

Notice that in the erroneous case, two user-defined conversions would have made it possible: first converting `1` to an `s` via constructor `s(int)`, and then converting `s` to an `s2` via the conversion function. But only one user-defined conversion is applied, and so this is erroneous.

User-defined conversion functions are inherited, and they can be virtual.

*Converting to a  
known type*

There are two possible contexts. The first is when a type you are converting to (“destination type”) is known, and the second is when no destination type is known. A destination type is known in the following contexts:

- an argument to a function (the destination type is the argument type)
- an initializer value (the destination type is the type of the object being initialized)
- a function return value (the destination type is the return type)
- an explicit type conversion (the destination type is the cast type)

These rules apply when a destination type is known are:

1. Apply user-defined conversions first to see if you can convert exactly to the destination type.
2. If not, see if you can convert to the destination type, ignoring any **const** and **volatile** qualifiers.
3. If not, see if it is possible to convert to a type that can in turn be converted (without user-defined conversions) to the destination type.

For example:

```
struct s {
    operator int();
    operator const int();
};
s x;
int i = x;      // Converts via operator int()
const j = x;    // Converts via operator const int()
int k = int(x) > 1 ? ... : ... ; // Converts via
                                // operator int()
```

*Converting to an unknown type* A specific destination type is not known in the following contexts:

- an operand of an expression
- the *expression* in:
  - `if(expression)`
  - `switch(expression)`
  - `while(expression)`
  - `for (S expression; e2)`

In these cases, conversion is attempted to either an integral type, a floating-point type, a pointer type, or a combination of these types, depending on the context.

*Overloaded operators and user-defined conversions* Because a call to an overloaded operator can require a user-defined conversion, ambiguities can arise between user-defined operators and pre-defined C++ operators. For example:

```
struct s {
    operator +(int);
    operator int();
};
s x,y;
int z = x+y;      // Error: ambiguous
```

Either of the following solutions is valid:

```
s::operator +(x, int(y))
```

or:

```
int(x) + int(y)
```

Both employ user-defined conversions. The compiler responds with a diagnostic message.

*Conversion by constructor* Whenever conversion by constructor is considered, overload resolution is used to choose a constructor from among many. If overload resolution fails in finding a single constructor, conversion by constructor is not considered. For example:

```
struct s {
    s(int);  s(float);
};
s x = 1;      // Converts via s(int)
s y = 1.2f;   // Converts via s(float)
```

*Using C-style casts for conversion functions* Not all conversion functions can be called by the function-style syntax of casting; sometimes old C-style syntax is necessary:

```
struct s {
    operator int();
    operator const int();
};
s x;
int i = int(x);      // Converts via operator int()
int j = (const int)x; // Converts via
                      // operator const int()
```

So, although C++ discourages use of the C-style cast, it is useful in naming certain conversion functions. Alternatively, you could more clumsily use a **typedef**:

```
typedef const int CI;
struct s {
    operator int();
    operator CI();
};
s x;
int i = int(x);      // Converts via operator int()
int j = CI(x);       // Converts via operator CI()
```

## 11.4 Operators `new` and `delete`

Operators `new` and `delete` allow control over storage allocation.

*Creating objects with `new`* When an object is created with the `new` operator, an `operator new()` function is implicitly called to obtain the storage necessary. If the object being created is of some class type `C`, `C::operator new()` will be invoked if it exists. The usual access and overloading rules apply.

*Characteristics of `operator new`* `operator new`:

- must be declared with its first argument of type `size_t` (defined in the header `stddef.h`)
- must return `void*`
- can have any other parameters:

```
struct s {
    void * operator new(size_t amount);
    void * operator new(size_t amount, char *msg);
};
```

The first argument to `new` is the size of the object being allocated. Read about the `new` expression in §6.4.10.11: *The new Operator* on page 91 to see how other arguments are passed to `new`.

`operator new` should return the address of the storage allocated.

You might provide a class-specific `operator new` to efficiently zero out the storage allocated to the class:

```
void * s::operator new(size_t amount) {
    return calloc(1,amount);
}
```

*Destroying objects with `operator delete`* When an object is destroyed with the `delete` operator, an `operator delete()` function is implicitly called to free the storage as necessary. If the object being destroyed is of some class type `C`, `C::operator delete()` will (usually) be invoked if it exists. The usual access rules apply. Overloading is not necessary because there can be only one `operator delete()` for a single class.



Characteristics of **operator delete**

**operator delete**

- must be declared with its first argument of type **void \***
- can have an optional second argument of type **size\_t** (defined in the header **stddef.h**)
- must return **void**:

```
struct s {
    void operator delete(void *ptr);
    // OR:
    // void operator delete(void *ptr, size_t size);
};
```

We say “usually invoked” because what really happens depends on whether a destructor exists for class **C** and whether that destructor is **virtual** and over-ridden.

Calling **delete** when no destructor exists

Consider the simple case where no destructor exists for a class **C**, but **C::operator delete** is declared. Then the **delete** expression calls **C::operator delete** with the pointer to the object being deleted as the first value and, if **delete** is declared with two parameters, **sizeof(C)** is passed as the second argument.

```
s *p;
delete p;    // Calls s::operator delete(p)
```

Calling **delete** when there is a destructor

Consider the case where a destructor exists for class **C**. Recall that destructors must call the operator **delete** if they are told to free the storage associated with an object. Thus, for an object of class **C**, if **~C()** exists, the **delete** expression calls the destructor for that object, passing it the value **TRUE** for the hidden parameter **I\_should\_delete**. The destructor itself calls the operator **delete** defined in the class containing the destructor.

```
struct s {
    void * operator delete(void *ptr);
    ~s();
};
s *p;
delete p;    // Calls s::~~s() with TRUE
             // for I_should_delete
```

Here the **delete** expression does *not* directly call the **delete** operator. It calls the destructor for the class, telling the destructor to delete the object.

Calling the destructor instead means that if **C**'s destructor was overridden by a derived class **D**, it is **D**'s destructor and **D**'s **delete** (if any) that will be called; **C**'s **delete** will never be called:

```

struct s {
    void * operator delete(void *ptr);
    virtual ~s();
};
struct s2 : s {
    void * operator delete(void *ptr);
    ~s2() ;
s *p = new s2;    // Creates an s2, put pointer in p
delete p;         // Calls s2::~~s2() with TRUE
                  // for I_should_delete

```

In this example, the destructor is virtual and is overridden by **s2**'s destructor. Therefore **s2**'s operator **delete** is called, not **s**'s. This is indeed appropriate behavior, because the pointer originated from an allocation of **s2**, not from an allocation of **s**.

*new and delete are static member functions* Operators **new** and **delete** are implicitly static member functions, even if you do not declare them **static**. For **new**, this is because there is no object yet created to be passed as the **this** parameter. For **delete**, it is because the process of deletion invalidates the **this** parameter. Because the functions are **static**, they cannot be **virtual**.

---

## 11.5 Assignment Operator =

The assignment operator for a class **S**, **C::operator =**, is invoked whenever an assignment (=) is made into an object of type **C**.

### Example 11.3

---

```

struct str {
    operator =(char *str);
    operator =(str&);
    str() { string = 0; }
    char *string;
};

```

```

// Assigns string s into this->string, by value:
str& operator =(char *s) {
    if (string) delete string;
    string = new char[strlen(s)+1];
    strcpy(string,s);
    return *this;
}

str& operator =(str& other) {
    // Checks for self-assignment; if so, does nothing:
    if (&other == this) return *this;
    // Now calls operator =(char*) to finish the job:
    return *this = other.string;
}

str x, y,z;
void f() {
    x = y;           // x.str::operator=(y)
    x = "hi";        // x.str::operator=("hi")
    y = x = "fnc";   // y.str::operator=
                    //   (x.str::operator("fnc"))
}

```

In Example 11.3 on page 186, class `str` contains a string. The user-defined assignment operators can copy into that string from another string (`operator =(char *)`) or from another `str` object (`operator =(str&)`). In either case the copying is done by value — that is, the string content, rather than the pointer to the string, is copied.

`operator =(char *)` reclaims any storage in the `str` object being assigned, allocates enough space to hold the copy, and then makes the copy. `operator =(str&)` checks whether the object for which it is called and the argument are the same, in which case nothing is done; otherwise, it calls the other assignment operator to do the actual string copy.

*Assignment operators should return **\*this*** The assignment operators shown in Example 11.3 on page 186 return **\*this**. This is recommended practice, because it allows for expressions such as:

```
y = x = "fnc"
```

This is equivalent to:

```
y.operator=(x.operator=("fnc"))
```

and is what you would expect with right-to-left assignment.

*Special default assignment operator* A “default” assignment operator for a class **C** is one that can take a single argument of class **C**. Such an operator is the one invoked when an object of class **C** is assigned from another object of class **C** or class derived from **C**. The default assignment operator **=** is considered special: if you do not provide one, the compiler generates one (unless **C** is an anonymous union). The compound assignment operators **+=**, **-=**, **\*=**, and so forth, are not considered special in this way.

*Compiler-generated default assignment* If all bases and members of a class **C** have assignment operators accepting **const** arguments, the generated default assignment operator has a single parameter of type **const C&**. Otherwise, it has a single parameter of type **C&**, and therefore assigning from a **const C** object is not possible. The return type of the generated default assignment operator is **C&**, and the operator returns **\*this** as its result. The semantics of a generated default assignment operator is to copy each member and base via the appropriate assignment operator for the member or base. In practice, the compiler generates fast bit-wise copy semantics for this copying when there are no user-defined assignment operators involved.

In a manner similar to constructors, compiler-generated assignment code takes care of assigning bases and members, and recognizes whether it is the topmost assignment operator, as shown in the following pseudo-code:

### **Pseudo-code: Assigning Bases and Members**

---

```
assignment_code(boolean I_am_topmost) {
    0. if (I_am_topmost) {
        Initialize all virtual base pointers for all bases. For each base in
        a depth-first, left-to-right traversal of the class graph, assign
        each virtual base and pass FALSE as the value for
        I_am_topmost.
    }
    1. Call the assignment operator for each direct, non-virtual base of C
    in left-to-right order. Pass FALSE as the value for I_am_topmost.
```

2. Call the assignment operator for each member of `C`. Pass `TRUE` as the value for `I_am_topmost`. This includes the virtual-function pointer, since it is a (compiler-hidden) member of the object being copied.

```
}
```

---

For example:

```
struct s1 {
    operator =(s1&);
};
struct s2 {
    operator =(s2&);
};
struct s3 : s1, virtual s2 {
    s1 mem_s1;
}
void f() {
    s3 x, y;
    x = y;
}
```

The compiler-generated assignment operator for `x = y` first assigns virtual base `s2` using `s2::operator=()`, then assigns base `s1` using `s2::operator=()`. `mem_s1` is then assigned using `s1::operator=()`.

*Avoid user-defined  
assignment  
operators in classes  
with virtual bases*

Unfortunately, whereas compiler help in copying the contents of virtual (versus non-virtual) bases is essential in user-defined copy constructors, no such compiler help is available in user-defined assignment operators. (Perhaps an extension to C++ would allow the compiler to graft its own code onto user-defined assignment code.) Thus, the use of user-defined assignment operators in classes with virtual bases is hazardous and best avoided.



# Name Overloading and Operator Functions in C++

---

This chapter discusses overloading function names, and shows how overload resolution determines which function is to be called.

*Functions can be multiply declared in C++*

Unlike C, C++ allows multiple declarations for a function in the same scope. The function name is then said to be *overloaded*. When the function name is subsequently used, *overload resolution* selects one declaration from the many declarations.

---

## 12.1 Overload Resolution

C++ provides two kinds of overload resolution: argument match and function match.

*Argument-match overload resolution*

Argument-match overload resolution is by far the more common of the two overload resolution processes, and occurs in a function-call context. This matching chooses a function by comparing the types of the arguments passed with the parameter types for each candidate function declaration. The return type of the function is not considered during argument-match overload resolution.

*Function-match overload resolution*

Function-match overload resolution chooses a function by comparing the type of a given function with each candidate, until an exact type match is found. The following example illustrates both kinds of resolution.

```
void f(int);
void f(float);
void f(char *);
// Argument-match overload resolution:
void g() {
    f(1);           // Calls f(int).
```

```

    f(1.2f);    // Calls f(float)
    f("str");   // Calls f(char *)
}
// Function-match overload resolution:
void (*h1)(float) = f; // f(float) chosen
void (*h2)(int)   = f; // f(int) chosen
void (*h3)(char *) = f; // f(char *) chosen
void (*h4)(double) = f; // Error: no exact match

```

*Parameter types can vary* Two function declarations by the same name are allowed when the parameter types are not identical; the return types do not matter:

```

void f(int);
int f(int); // Not valid: same parameter
            // types as previous f

```

Some functions with different parameter types cannot be distinguished from each other in argument-match overload resolution because they accept the same parameter types. For example:

```

void f(int);
void f(int &); // Parameter types differ, so
              // allowed

void g() {
    f(1); // Cannot tell which f;
          // both take an int
}

void (*h)(int &) = f; // Selects f(int &)

```

However, function-match overload resolution *can* tell the difference.

The reason the return type is not considered in argument-match overload resolution is that using the return type would make it harder for compilers — and much harder for programmers — to recognize which function is chosen by overload resolution.

---

## 12.1.1 Argument-Match Overload Resolution

*Choosing the “best” function* When you call a given function with a given set of arguments, argument-match overload resolution chooses the “best” function from among all candidate functions that can possibly be called. A function is best unless it is “bested” by another function. A function is bested if, for any argument



expression, another function has a better “score” on matching the expression’s type to that of the parameter type of the function. There must be exactly one best function, or else the call is not valid. Two kinds of error can result:

- no best function
- multiple best functions

### Example 12.1

---

```
void f(double);
void f(float);
void g(double,int);
void g(int,double);
void test() {
    f(1);          // Error: there are two "best" functions,
                   // f(double) and f(float); both require
                   // the same conversion from 1 to a
                   // floating-point number.

    f(1.2f);       // f(double) has a worse score than
                   // f(float), because converting 1.2f to
                   // double is worse than not converting
                   // it at all; hence f(float) wins.

    f(1.2);        // f(float) has a worse score than
                   // f(double) because converting 1.2 to
                   // float is worse than not converting
                   // it at all; hence f(double) wins.

    g(1,1);        // Error: there are no "best" functions.
                   // g(double,int) is bested by
                   // g(int,double) when considering the
                   // first argument, and vice-versa for
                   // the second argument.

    g(1,1.2);      // g(int,double) scores better than
                   // g(double,int) for both arguments.
}
```

---

## 12.1.2 Argument Type Matching

The type **T** of each argument expression is examined in turn and compared with the corresponding parameter type **A** of each candidate function. Each argument receives a score as follows:

Score 0: (Best possible score.) If **T** equals **A**, or **T** is converted to **A** by any of the following trivial conversions on a type **X**:

```
X  => X &
X  => const X
X  => volatile X
X  => const volatile X
```

In other words, converting a type by adding **const** or **volatile** or by turning it into a reference type is the best possible conversion, if a conversion is necessary.

Score 1: If the conversion from **T** to **A** requires any of the conversions for score 0, plus:

```
X & => const X &
X & => volatile X &
X & => const volatile X &
X * => const X *
X * => volatile X *
X * => const volatile X *
```

In other words, adding **const** or **volatile** to the base type of a pointer or reference type is worse than score 0 (zero).

Each of the higher scores below specifies what more is permissible in addition to what is allowed by previous scores.

Score 2: Add integral promotions or conversions from **float** to **double** (see §6.5: *Conversions* on page 93).

Score 3: Add standard conversions (see §6.5: *Conversions*), excluding those standard conversions mentioned in Scores 4 and 5.

Score 4: Add conversion from **C\*** to **B\*** or from **C&** to **B&**, where **B** is a publicly derived base of class **C**.

Some conversions in this category can be better than others, but a simple scoring mechanism cannot be used to compare them.

The conversion `C*` to `B*` is better than `C*` to `A*` if `A` is a base of `B`; essentially, `B` is “closer” to `C` than is `A` so `C*` to `B*` is preferred over `C*` to `A*`. However, if `A` is not a base of `B`, both `C*` to `B*` and `C*` to `A*` have the same rank and neither is preferred. The same argument holds for `C&` to `B&` versus `C&` to `A&`.

Score 5: Add conversion of `X*` to one of the following:

```
void *
const void *
volatile void *
const volatile void *
```

Score 6: Add a single user-defined conversion.

Score 7: If there is no parameter `A`, but there is an ellipsis in the function, so that `T` is accepted because the function declaration contains “...”.

No sequence of conversions is considered if it is longer than another sequence that achieves the same effect. For example, `int->float->double` is not considered because `int->double` achieves the same effect and is shorter.

For the purposes of scoring, a function with `n` trailing default parameters is considered to be `n+1` functions with differing numbers of parameters. For example:

```
void f(int, char = 'c', float = 1.2);
```

is considered to be these three candidates:

```
void f(int);
void f(int, char);
void f(int, char, float);
```

and so can match `f(1)`, `f(1, 'x')`, and `f(1, 'x', 3.3)`.

For the purposes of scoring, a non-static member function is considered to have an extra first parameter type — the type pointed to by the `this` pointer. In converting the argument, no temporaries are introduced, nor are any user-defined conversions attempted to achieve a type match.

For the purposes of scoring, an ellipsis (...) in the function declaration matches any sequence of argument types, each then having the ellipsis score as already described.

```

void f(double);
void f(float);
void g(double,int);
void g(int,double);
void test() {
    f(1);           // Error: Both functions score 3 on
                   // converting 1 to double or float.

    f(1.2f);        // f(double) scores 2 and f(float)
                   // scores 0 on the argument, so
                   // f(float) wins.

    f(1.2);         // f(float) scores 2 and f(double)
                   // scores 0 on the argument,
                   // so f(double) wins.

    g(1,1);         // Error: g(double,int) scores (3,1)
                   // and g(int,double) scores (1,3).
                   // So each bests the other in
                   // one of the arguments.

    g(1,1.2);       // g(int,double) scores (1,2)
                   // and g(double,int) scores (3,3),
                   // so g(int,double) clearly wins.
}

```

---

### 12.1.3 Function-Match Overload Resolution

*Choosing a matching function type* In function-match overload resolution, a function name is used without any arguments, and it is converted to another function type. Function-match overload resolution chooses the declaration of the name that exactly matches that other function type.

Function-match overload resolution occurs in these contexts:

- a function pointer being initialized
- a function pointer on the left side of an assignment
- an argument in a function call where the corresponding parameter is a function pointer
- a function return expression having a function pointer type

If an exact match is not possible, function-match overload resolution fails with no solution.

```
void f(int);
void f(float);
void f(char *);
// Examples of function-match overload resolution:
// typedef for simplicity
typedef void (*func_taking_int)(int);
func_taking_int test() {
    void (*h1)(float) = f; // f(float) chosen
    void (*h2)(char *);
    h2 = f;                // f(char *) chosen
    extern void g( void (*)(int) );
    g(f);                  // f(int) chosen
    return f;              // f(int) chosen as return result
}
```

---

## 12.2 Overloading and Scope

*Functions must be in the same scope* C++ allows multiple declarations for a function in the same scope. However, C++ does not allow a function in one scope to overload a function in another scope. For example:

```
void f(int);
test() {
    extern void f(char *);
    f("hi"); // OK, calls f(char*).
    f(1);    // Error: no function callable
}
```

At the call `f(1)`, there is just one candidate function: `f(char*)`. The global `f(int)` is hidden, not overloaded by `f(char*)`.

## 12.3 Operator Functions

*Defining your own operations* In C++ you can declare functions that can allow the language's built-in operators such as `+` to work on types that you define. These are called *operator functions*. The name of an operator function has the form **operator** followed by one of the C++ operators in the following list.

<b>new</b>	<b>delete</b>					
<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>++</code>	<code>--</code>
<code>^</code>	<code>&amp;</code>	<code> </code>	<code>~</code>	<code>&lt;&lt;</code>	<code>&gt;&gt;</code>	
<code>=</code>	<code>+=</code>	<code>-=</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>	
<code>^=</code>	<code>~=</code>	<code> =</code>	<code>!=</code>	<code>&gt;&gt;=</code>	<code>&lt;&lt;=</code>	
<code>&lt;</code>	<code>&gt;</code>	<code>==</code>	<code>&lt;=</code>	<code>&gt;=</code>		
<code>&amp;&amp;</code>	<code>  </code>	<code>!</code>				
<code>,</code>	<code>-&gt;*</code>	<code>-&gt;</code>	<code>()</code>	<code>[]</code>		

The following operators *cannot* be declared:

`.`      `.*`      `::`      `:::`      **`sizeof`**

Both the unary and binary forms of `+`, `-`, `*`, and `&` can be declared; whether a function is a unary or binary operator depends on whether it takes one or two arguments. For example:

```
class s {
    int operator -(int); // Binary: Remember the
                        // hidden "this" parameter.
    int operator &();    // Unary: The argument is the
                        // hidden "this" parameter.
};
int operator+(int,s&); // Binary
double operator *(s&); // Unary
```

An operator function permits defining, say, the operation `+` on a string class — perhaps to indicate string concatenation:

```

string s1(80), s2(80);
string operator + (string &parm1, string &parm2);
. . .
string s3(160) = s1 + s2;

```

*Declaring an  
operator function*

An operator function can be declared in either of two ways:

- as a non-static class member function
- as any other declaration (that is, *not* within a class), provided at least one of its parameters is a class type or a reference to a class type

The distinction is due to the way an operator function is chosen in an expression. When you write a binary operator expression such as:

```
x + y
```

the C++ built-in binary addition operator is used if neither argument is a class type. Otherwise, the built-in operator cannot possibly apply, and so the following two scopes are examined for a declaration of an operator function to be called:

- the nearest scope that is not a class scope
- the class scope where `x`'s type was declared, if `x` is a class object

*Overload resolution  
chooses between  
scopes*

If there is an operator in both scopes, overload resolution is used to select which operator. Even though the operator function might not be overloaded in either scope, overload resolution must be used because candidates come from two different scopes.

Similarly, when you write a unary operator expression — for example, `+x` — the same two scopes are considered.

*Overloading  
operator functions*

Just as for non-operator functions, you can also overload operator functions. In general, the overloaded operators in both scopes are candidates from which overload resolution chooses a single operator.

Because operator functions are just functions with special names, you can still call them with the same syntax you use to call any function. In fact, you can use the operator function name just as you would use any other function name:

```

operator+(x,y);
x.operator+(y);    // operator+ must be
                   // a member function.

```

```
int (s::*add)(s&) = x.operator+;
g(operator+);      // Passes operator+ as an argument
```

*Calling operator functions implicitly*

The only thing special about an operator function is that it can be called implicitly in an expression such as `x + y`. Essentially, `x + y` is equivalent to either `operator +(x, y)` or `x.operator+(y)`, depending on where the operator function is found. You could write `operator +(x, y)` or `x.operator+(y)` yourself, but neither means the same thing as writing `x + y`, where the compiler chooses between the two.

In Example 12.2, `operator*` is overloaded in the scope of the `complex` class and in the global scope.

### Example 12.2

---

```
struct complex {
    complex operator +(int);           // Allows
                                      // complex+int
    complex operator +(complex);      // Allows complex
                                      // + complex
    complex(double,double);          // Constructor
};
complex operator *(complex&,int)      // Allows
                                      // complex*int
complex operator *(complex&,double); // Allows
                                      // complex*double
complex operator *(int,complex&);    // Allows
                                      // int+complex

test() {
    complex c1(1, 3), c2(-1, 6);
    complex c3;
    c3 = c1 + c2; // Equivalent to c1.operator+(c2)
    c3 = c1 + 1;  // Equivalent to c1.operator+(1)
    c3 = c1 * 2.7; // Equivalent to operator*(c1, 2.7)
    c3 = c2 * 1;  // Equivalent to operator*(c1, 1)

    c3 = 1 + c1;  // Equivalent to operator+(1, c1)
    c3 = 1 * c2;  // Error: no operator found
}
```

---



*Operator functions versus built-in operators* The usual properties that apply to built-in operators do not necessarily apply to operator functions. For example, `1 + x = x + 1` for built-in `+`. However, as shown in Example 12.2, `c1 + 1` and `1 + c1` call different functions. And although you could write `c2 * 1`, no operator function applied for `1 * c2`. `++x` is the same as `x += 1` for built-in `++` and `+=`, but this might not hold for operator functions `++` and `+=`.

*Where to place an operator function* You have two choices of position for an operator function. You would normally place an operator function within a class if one of the following conditions applies:

- The function requires access to the private data of the class.
- The function would be more efficient if it could assume details of the class representation that only member functions would know.
- The function is one of `operator=`, `operator[ ]`, `operator( )`, or `operator->`. These must be non-static member functions.

---

### 12.3.1 Binary Operators

The expression `x OP y` is interpreted either as `x.operator OP(y)` or as `operator OP(x, y)`, with overload resolution choosing among possible candidates. The binary `operator =` is defined for a class, unless you define it. See §11.5: *Assignment Operator =* on page 186 for more information.

---

### 12.3.2 Prefix Unary Operators

`OP x` is interpreted as either `x.operator OP( )` or `operator OP(x)`, with overload resolution choosing among the candidates.

---

### 12.3.3 Postfix Unary Operators `++` and `--`

Operators `++` and `--` can be both prefix and postfix, and so merit special attention. Postfix operator functions `++` and `--` are declared with two param-

eters, the second of which is type **int**. When called, 0 (zero) is passed as the second argument:

```
class s { ... };
operator ++(s&);           // Prefix
operator ++(s&, int);      // Postfix
s x;
++x;                       // Calls prefix
x++;                       // Calls postfix
```

---

### 12.3.4 Postfix Function-Call Operator ( )

The function call:

```
function_expression ( arguments )
```

is considered an  $n$ -ary operator with *function\_expression* as the first operand and the arguments as the remaining operands, for a total of  $n$  operands. **operator()** must be a non-**static** member function, so the *function\_expression* can be only of a class type.

*function\_expression*  
( *arguments* ) is equivalent to:

```
function_expression.operator() ( arguments )
```

*Multi-dimensional arrays* You could use **operator()** to implement a multi-dimensional array:

```
class _3Darray {           // Of ints, say
    int *storage;
    int one, two, three;   // Dimensions
    int& operator()(int x, int y, int z) {
        return storage[x*two*three + y*three + z];
    }
};
_3Darray x;
...
x(1, 2, 3) = 1;           // Calls x.operator()(1, 2, 3).
```

### 12.3.5 Postfix Subscript Operator [ ]

Operator [ ] is considered a binary operator. `x[y]` is interpreted as `x.operator[](y)`. `operator[]` must be a non-static member function.

*Associative arrays* You could use `operator[]` to implement an associative array:

```
class AA {
    int operator[](int);
};
AA x;
. . .
x[1] = 1;
x[4000] = 2;
// At this point there might be only two objects
// stored, at subscripts 1 and 4,000.
```

### 12.3.6 Postfix Dereference Operator ->

Operator -> is considered a unary operator. `x->y` is interpreted as `(x.operator->())->y`. `operator->()` must be a non-static member function. `operator->` must therefore return something that can be used as a pointer.

*Smart pointers* You could use `operator->` to implement so-called “smart pointers”; that is, pointers that do something extra when dereferenced, such as make sure that the dereferenced area is paged in from secondary storage. For example:

```
class object {    // That is paged to disk
    int field;
};
class smart_ptr {
    object *p;
    long disk_location;
    object * operator->() {
        // Make sure x.p is in memory.
        . . .
        return p;
    }
};
```

```
smart_ptr P;  
.  
.  
.  
int i = P->field;
```

Make sure **operator->** does not return the same type that it takes:

```
class loop { loop operator->(); int field; };  
loop x;  
x->field;  
// Means (x->operator->())->field, which means  
// ((x->operator())->operator())->field, which means  
// (((x->operator())->operator())->  
// operator())->field, . . .
```

# 13

## C++ Templates

---

*Families of types or functions* A **template** defines a family of types or functions. Using templates, you can declare a **class** whose members can assume various types, based on parametric substitution, or you can define a family of functions, whose argument types and local variables can similarly assume various types.

*Declaring a template* This is the form of a **template** declaration:

```
template < template_arg[ , template_arg]... >  
    declaration
```

where *template\_arg* is one of the following:

```
class identifier  
argument_declaration
```

See Chapter 8: *Functions* for the definition of an *argument\_declaration*.

The **class identifier** form of *template\_arg* means that the corresponding argument must be a type expression (actually, an *abstract\_declarator*) and in the body of the template *declaration* the identifier stands for that argument type.

*declaration* is a declaration of a function or a class.

Example 13.1 defines a class template. **tree** is a template class where type **node** can assume any type, forming a class object different from any other class object defined using this template. For example, **oak** is an array of 20 class objects, and within each class object defined for **oak**, elements having the type **node** in the template declaration assume the type **int**. But **pine** is an array of 30 class objects whose **node** members assume the type **unsigned**.

**Example 13.1**


---

```

template <class node> class tree {
    node *left_child, *right_child;
    int size;
public:
    tree();
    node& operator[] (int);
    node& element(int i) { return left_child[i]; }
};
tree<int> oak[20];
tree<unsigned> pine[30];
tree<unsigned> pinon[10]; // Same element type as pine

```

---

As Example 13.1 shows, this is the syntax for using a class template in a declaration:

```

template_name < template_actual
    [, template_actual] ... >

```

where *template\_name* is the name of the class declared in a template's *declaration*.

*Template arguments*

Templates have fundamentally two kinds of arguments: types and expressions. If the *template\_arg* is of the form *class identifier*, the argument must be a type. Otherwise, the argument must be an expression. So, the *template\_actual* can be one of the following:

- a type:
  - an *abstract\_declarator* denoting a fundamental or derived type
- a limited form of expression:
  - a constant expression
  - the address of an object with external linkage
  - the address of a static class member

Rules for declaring  
a class **template**

The following rules apply to class **template** declarations.

- A **template** declaration can appear only at file scope.
- A class template name must be unique in a program. It cannot refer to any other template, class, function, object, value, or type in the same scope.
- For a non-type template argument, the type of a *template\_actual* must match that given in the template declaration for that argument.
- You can use the *template\_name* only by instantiating it.

---

## 13.1 Template Type Equivalence

Two instantiations of a template refer to the same type if the names are the same, and if their argument lists are the same or differ only in constant expressions that evaluate to the same value.

You can declare objects using a class template many times, and the compiler makes sure that only one instance for each set of distinct arguments is generated.

```
template< class mytemp,
          int size1,
          int size2 > class mybuf {
    mytemp *buf1;
    buf1size[size1];
    buf2size[size2];
    mytemp *buf2;
public:
    mybuf();
}
mybuf<char,120,80> line1;
mybuf<char,3*40,160/2> line2;
```

This example declares a **template** and two objects that use it. The argument lists for the two objects `line1` and `line2` are different, but they both use the same **class** name, and their argument lists contain constant expressions that evaluate to the same values. `line1` is therefore the same type as `line2`.

## 13.2 Function Templates

*Declaring a family of functions* A function **template** allows you to declare a family of logically related but separate functions. These functions can perform the same action, but operate on different data structures. You declare a function template when the *declaration* portion of a **template** declaration is a normal C++ function declaration.

For example, suppose you define a class that implements a binary-tree data structure. Each class object contains a pointer to a left child node and a right child node, as in Example 13.1. You could define a family of functions to manipulate the tree, as shown in Example 13.2.

### Example 13.2

```
tree<int> oak, willow;
tree<float> spruce;
template <class node> void treeop(tree<node> N) {
    node *p1, *p2;
    ...
}
void func() {
    treeop(oak);           // Calls treeop(tree<int>),
                          // passing oak
    treeop(spruce);        // Calls treeop(tree<float>),
                          // passing spruce
    treeop(willow);        // Calls treeop(tree<int>),
                          // passing willow
    void (*fp)(tree<float>) = treeop;
    // Finds the function created by the call to
    // treeop(spruce) and assigns it to fp
    fp(spruce);           // Same as treeop(spruce)
    void (*fp2)(tree<long>) = treeop;
    // Instantiates treeop(tree<long>) and
    // assigns it to fp2
}
```



The template definition of `treeop` in Example 13.2 declares an unbounded set of functions, each named `treeop`. When `treeop(oak)` is seen, the compiler provides a definition for a `treeop` function that can take the type of oak:

```
void treeop(tree<int> N) { int *p1, *p2; ... }
```

Notice how the parameter `node` has assumed the type `int`. The compiler inferred this because the formal parameter was `tree<node>` and the actual parameter was `tree<int>`.

Later, when `treeop(spruce)` is encountered, the compiler provides a similar definition:

```
void treeop(tree<float> N) { float *p1, *p2; ... };
```

When `treeop(willow)` is encountered, the compiler has already found a `treeop` function that can be called with `tree<int>`, and so does not provide another declaration.

To initialize function pointer `fp`, the compiler uses `treeop<float>`, which is already defined. To initialize `fp2`, the compiler instantiates a `treeop<long>` function whose address is assigned to `fp2`.

*How the compiler chooses a function*

The presence of templates modifies the compiler's behavior in searching for functions to call or to take the address of. Here is what happens, in order:

1. The compiler tries to find a function that can be called without any type conversions — this is called an “exact match”.
2. If there is no exact match, the compiler looks for a function template that can be instantiated and which then provides a function that can be called with an exact match.
3. If no such function template is found, the compiler performs a normal function search as if there were no template functions. This may involve overload resolution, argument conversion, and so forth.

So, by the time the compiler encounters the call `treeop(willow)` (for example) in Example 13.2, a function already exists (by previous template instantiation at the call to `treeop(oak)`) that could be called with an exact match, so the compiler does not instantiate another function definition.

*Rules may change*

---

**Note:** These three rules are a subject of controversy in the ANSI X3J16 committee for standardization of C++. Many want the “exact match” rule to be weakened slightly. Expect the rules to

be changed, at least slightly. Consult the [README](#) file in the compiler distribution for information about any such changes implemented in High C++.

---

Given these rules, in Example 13.2 you could provide your own definition of a “specialized” `treeop` function, in case you do not like what would be instantiated from the template definition:

```
void treeop(tree<double> N) {
    double D;
    ...
}
tree<double> apple;
tree<char *> ibm;
void func() {
    treeop(apple);      // Calls non-template
                        // treeop(tree<double>)
    treeop(ibm);        // Calls template
                        // treeop(tree<char *>)
}
```

Here the specific definition of `treeop(tree<double>)` causes the compiler to find this function first at the call to `treeop(apple)` (by the “exact match” rule) and so there is no need to instantiate one from the template definition. This works even if your definition appears *after* the call to `treeop(apple)`. You might use specific definition of functions to write a faster version of the general template function for certain argument types.

*Declaring a  
function object  
more than once*

You can declare function objects using a function template many times, and the compiler makes sure that only one instance for each set of distinct template arguments is generated. You declare a function template object when you call a function **template** or take its address.

---

## 13.3 Template Declarations and Definitions

A class or function template can have exactly one definition for each **template** with a particular name in a program. That definition is provided by the compiler instantiating a class or function template, or by you providing a specific definition of a class or function template.

*Overriding a template function* If you define a function that is not a **template** function, and its type and parameters exactly match an instantiation of a **template** function with the same name, the definition of the non-template function is a definition of the matching **template** function.

In a similar way you can provide a specific class definition to override a template instantiation:

```
class mybuf< char *, 100, 200> {
    ...
};
mybuf<char*,100,200> line3;
```

Here the type of `line3` does not cause a template instantiation of `mybuf` (see Example 13.1 on page 206), but instead uses the specific class you have provided. You might use a specific class definition to obtain efficiency (for some template arguments) greater than would be afforded by instantiation of the general template.

---

## 13.4 Template Member Functions

When you declare a **template** class, any member function of the class is implicitly a **template** function. Each such member function has the template arguments of the class of which it is a member as its own template arguments.

### Example 13.3

---

```
template <class node> class tree {
    node *left_child, *right_child;
    int size;
public:
    tree();
    node& operator[](int);
    node& element(int i) { return left_child[i]; }
}
tree<int> oak(20);
// The implicit template function, which has
// the same template arguments as class tree:
```

```

template <class node> node& tree<node>::
    operator [] (int i) {
        if ( i < 0 || size <= i )
            cout << "Tree: subscript error.";
        return left_child[i];
    }

```

---

In Example 13.3, a subscript member function `operator [] (int)` is defined. It has the same arguments as its `template` class declaration.

Notice, however, the syntax for saying that you are defining a member function of a template class: `tree<node>::operator [] (int i)`. The `tree<node>::` specifies that the function being defined is a member of the general `tree` template.

---

## 13.5 Template Friends

A function that is a friend of a `template` class is not automatically a `template` function.

```

template <class job> class todo {
    friend void next_job();
    friend todo<job>* override(todo<job>*);
    friend todo* mistake(todo*); // Not valid
};

```

In this example, the function `next_job()` is a **friend** of all `todo` classes. In addition, each `todo` object has an appropriately typed function called `override()` as a **friend**. But the declaration of `mistake()` is erroneous, because the type `todo` does not exist. There are only `template` types, such as `todo<int>`.

---

## 13.6 Static Members and Variables in a `template`

Unlike normal classes, in a `template` class each generated class or function has its own copy of any **static** variables or members. This is because the members can assume different types.

For example:

```
template < class myclass > class uclass {  
    static myclass aname;  
}  
uclass<int> ii;  
uclass<char*> cc;
```

In this example, `ii` has a **static** class member `aname` that has type `int`. `cc` has a **static** class member `aname` that has type `char*`.



# C++ Namespaces

---

Namespaces are a recent addition to the C++ language. They were accepted in August 1993 at the Munich meeting of X3J16. Namespaces have been preceded by many such experiments in programming language design, such as Ada packages (1980), which were also incorporated in MetaWare's Professional Pascal (1984).

C++ namespaces offer new dimensions not present in Ada and Pascal, such as the interesting distinction between a **using** declaration and a **using** directive (see §14.3: *Using the Names Declared in a Namespace*).

---

## 14.1 When to Use Namespaces

The global namespace that C++ inherited from C causes a problem: global names of functions, variables, types, enumerators, and so forth declared in one third-party library may clash with names in another third-party library, or in your own application.

For example, a graphics library from vendor A might have a method called `Curve::rotate(int degrees)`, and so might the math library from vendor B. Your only recourse is to change the name of the class `Curve` in one library or the other. Sometimes the change is difficult or impossible to make.

C++ namespaces help solve this problem.

*Encapsulate library  
names*

Using namespaces, you can encapsulate library names in a library namespace with a long name derived, say, from the name of the library vendor. Vendor A's `rotate` function is then known as:

```
A_graphics::Curve::rotate
```

and vendor B's as:

```
B_math::Curve::rotate
```

The external (mangled) names given to the linker for such encapsulated library names includes the namespace name, so each name is distinct to the linker.

*Encapsulate names in your own code* Even if you are using a library that pre-dates namespaces, you can use namespaces in your own code to encapsulate your own identifiers that clash with those of the library.

---

## 14.2 How to Specify a Namespace

Names are introduced into a namespace by means of a **namespace** definition:

```
namespace identifier {
    ... list of zero or more declarations
}
```

*Namespaces are global* A **namespace** definition is syntactically a declaration, but it can appear only at the global scope or within another **namespace** definition. All names declared directly within a **namespace** definition, except those introduced by a **using declaration** (see §14.3.1: *The using Declaration* on page 218), are “owned” by the namespace.

*Namespaces can be extended* If **identifier** is not declared in the current scope, it is declared as a namespace. If it is already declared, it must be declared as a namespace, and the new **namespace** definition “continues” the namespace. In this way, a namespace can be extended:

```
namespace A {                      // Declares A as a namespace
    int f() { ... }
    typedef int T;
}

...
namespace A {                      // Adds more to namespace A
    int g();
}
```

This continuation allows a single large namespace to be defined over multiple header files.



A library vendor can choose a very long name for a namespace to minimize potential clashes with other vendors:

```
namespace Distributed_System_Object_Model_Library {
    ...
}
```

*Namespaces can be aliased*

Because it is unwieldy to use such a long name to refer to names in the namespace, you might want to use a namespace aliasing definition to create a shorthand name:

```
namespace DSOML =
    Distributed_System_Object_Model_Library;
```

Now `DSOML` is declared as a namespace, and it stands for the longer name in the scope of the alias definition.

---

## 14.3 Using the Names Declared in a Namespace

*Explicit qualification*

You can explicitly denote the names declared in a namespace by means of the existing qualified name syntax for classes. For example, `Colors::invert` denotes member `invert` of namespace `Colors`:

```
namespace Colors {
    enum color {red, white, blue};
    typedef int color;
    color invert(color);
    extern color c;
};

...
func() {
    Colors::color C = ...;
    Colors::invert(C);
}
```

A name declared in a namespace can be defined outside the namespace, if it is not already defined within the namespace. You use the qualified name to denote the member being defined:

```
Colors::color Colors::invert(color c) { ... }
Colors::color Colors::c = red;
```

Just as with classes, the text following the qualification is considered to be in the scope of the namespace. So, although you must use `Colors::color` for the return type, namespace members `color` (the parameter type) and `red` (the initializer) can appear without qualification.

### 14.3.1 The `using` Declaration

A **`using`** declaration declares a name *Name* from a namespace as a current declaration in the scope in which the **`using`** declaration appears. The declaration of *Name* is an alias of its declaration in the namespace. A **`using`** declaration uses a qualified name to refer to *Name*. For example:

```
func() {
    using Colors::red, Colors::white, Colors::blue;
    Colors::color x = red;
    using Colors::invert;
    x = invert(x);
}
```

---

**Note:** Although MetaWare High C++ allows the comma-separated list shown in this example, as of this writing the ANSI X3J16 draft standard does not.

---

A **`using`** declaration can overload functions

Just as with normal declarations, a **`using`** declaration can introduce a function that overloads other functions in the same scope. And just as with normal declarations, a **`using`** declaration can cause a duplicate declaration if a non-function of the same name has already been declared:

```
func() {
    extern int invert(int);
    using Colors::invert; // invert is now overloaded.
    // OK if invert is a function
    using Other_namespace::invert;
    Colors::color C = ...;
    invert(C);           // 1: Selects from 3 inverts.
    typedef int T;
    using Other_namespace::T; // Error!
}
```

Unlike normal declarations, **`using`** declarations can overload functions with the same argument types. The resulting ambiguity is detected at the point of

use, rather than at the point of declaration. For example, if `Other_namespace::invert` has the same signature as `Colors::invert`, then at `1:`, when the call is made to `invert(C)`, an error message results.

*Non-**using** declarations take precedence over **using** declarations*

If there is an ambiguity between two functions with identical argument types, where one function is introduced by a **using** declaration and the other by a non-**using** declaration, the non-**using** declaration takes precedence over the **using** declaration.

```
extern int invert(Colors::color);
using Colors::invert;    // invert is now overloaded.
Colors::color C = ...;
invert(C);               // 2: Selects from 2 inverts.
```

Here, at `2:` the call is made to the **extern int** function rather than to the function declared with the **using** declaration. No diagnostic is produced, and this silent choice could easily become a point of disagreement among language designers.

You can place a **using** declaration anywhere any other declaration can appear. A **using** declaration within a namespace *N*, just like any other declaration in *N*, causes the name of the identifier being declared to be declared in *N* as well, so *N::name* can refer to the **using**-declared name. However, the imported name is still “owned” by its original namespace, so being **using**-declared does not affect its external mangled name. For example:

```
namespace TV {
    using Colors::color;
    color pixels[1024*1024];
}
```

You can now write `TV::color` or `using TV::color`, even though `color` is owned by namespace `Colors`.

Partly because **using** declarations allow a name to be declared in multiple namespaces, and partly because **using** declarations are just aliases, distinct **using** declarations can import the same declaration. Such an otherwise invalid duplicate declaration is allowed with **using**, and has the same effect as if the duplicate were not there.

```
using Colors::color;
using TV::color;      // OK; same as Colors::color
```

This duplicate declaration is more useful with functions, where you can use a namespace solely for constructing a collection of functions:

```
namespace Collection1 {
    using A::f, B::f, C::f;
};
namespace Collection2 {
    using B::f, C::f, D::f;
};
...
... Collection1::f(...) ...
... Collection2::f(...) ...
```

Now you can also import both collections without fear of duplication:

```
using Collection1::f, Collection2::f;
... f(...) ... // Selects from A::f, B::f,
                // C::f, and D::f.
```

A **using** declaration does *not* allow you to select a single, individual function:

```
using N::f(int);           // NOT ALLOWED
using N::f(char);         // NOT ALLOWED
```

**using N::\*** High C/C++ also supports **using N::\***, which effectively performs a **using** on every identifier declared in *N*.

---

### 14.3.2 The **using** Directive

A **using directive** is quite different from a **using** declaration. The directive imports an entire namespace at once, but not as declarations in the scope containing the directive. Instead, within the scope containing the directive, the names are treated as if they were declared in the smallest enclosing non-namespace scope. (This is the global scope for namespaces declared globally or within global namespaces.) Furthermore, apparent duplicate declarations resulting from such treatment do not occur until the point of use of a name.

A **using** directive begins with **using namespace**, followed by the (potentially qualified) name denoting the namespace.

```
func() {
    using namespace Colors;
    color C = red;
    invert(C);
}
```

Because the names are treated as if they were declared globally, local declarations of the same name override global declarations:

```
func() {
    using namespace Colors;
    using Other_namespace::invert; // 1: Local
                                    // declaration

    color C = red;
    invert(C);                      // 2: Calls 1:
}
```

The local declaration at **1:** takes precedence over the global declaration, so the function call at **2:** invokes the function declared at **1:**. However, consider the following situation:

```
func() {
    using namespace Colors; // 1: invert declared
                            // globally
    using namespace Other_namespace; // 2: Another
                                    // global invert

    color C = red;
    invert(C);                    // 3: Chooses between
                                    invert at 1: and
                                    invert at 2:
}
```

Here, the members of both namespaces are introduced globally; therefore, at **3:** there is the choice of two globally declared `invert()` functions.

As with `using` declarations, ambiguity between two functions with identical argument types is resolved in favor of a function not introduced by a `using` definition. Likewise, ambiguity that cannot be resolved by overload resolution is reported at the use of the name, not at the occurrence of the `using` directive.

Names introduced by a **using** directive are ignored when an expression uses explicit qualification. Thus, even though the names are treated as if they were declared globally, a reference `::global_name` does *not* refer to any names introduced by a **using** directive:

```
namespace A { int glob; }
using namespace A;
int glob;      // Two globs declared globally
func() {
    glob++;    // Ambiguous: A::glob or ::glob?
    ::glob++;  // Refers to global, non-namespace glob
    A::glob++; // Refers to namespace A's glob
}
```

---

### 14.3.3 Namespaces, **friends** and **externs**

**friends** first declared in a class are declared in the smallest enclosing non-class, non-function-prototype scope. With the addition of namespaces, this means a **friend** declaration can be “injected” into the namespace:

```
namespace A {
    class C {
        friend func(); // This is A::func.
    };
}
int A::func() { ... }
```

The same holds for an **extern** declaration first declared within a function in a namespace. Rather than being a global **extern**, the declaration is owned by the namespace.

---

### 14.3.4 Unnamed Namespaces

If you omit the identifier in a namespace definition, you are referring to an “unnamed” namespace. Each compilation unit has one of these unnamed namespaces, which is unique for that compilation unit, and a **using** directive for it is automatically assumed. The effect is that you can enclose your local data in the unnamed namespace without fear of it clashing with local data in other compilation units.

For example:

```
namespace {
    int a, b, c;
    void f() { ... }
}
```

This is equivalent to the following:

```
namespace UNIQUE {
    int a, b, c;
    void f() { ... }
}; using namespace UNIQUE;
```

where `UNIQUE` is a compiler-chosen name unique for each compilation unit.

---

## 14.4 Current Status of Namespaces

As of the writing of this document, some issues are under discussion by ANSI Committee X3J16:

- the statement “the scope of a class is a namespace”
- the implicit internal linkage of unnamed namespaces
- the transitivity of directives
- the possible inclusion of `using N::*`

*using directives  
are not transitive*

According to a preliminary paper on namespaces (93-055), `using` directives could be transitive; that is, if namespace `A` contains `using namespace B` in its definition, `using namespace A` also provides access to the names in `B`. However, 93-105, the formal proposal for namespaces, does not indicate such an intent. What appears to be transitivity was intended for joining two namespaces together in a single namespace. High C++, to preserve the intent of 93-105, does not allow directives to be transitive.

*Linkage of names in  
unnamed  
namespaces*

The proposed C++ draft also says that names within the unnamed namespace have internal linkage, but this is still under discussion.

*Classes are not  
namespaces*

MetaWare’s implementation does not regard classes as namespaces, and does not use internal linkage for names in unnamed namespaces. It is not necessary to preserve clashes among different compilation units because, as one committee member observed, unnamed namespaces are unique, so none of the contained identifiers can ever clash when a program is linked.

*Templates cannot be namespace members* MetaWare's implementation does not allow templates to be namespace members.  
We will change MetaWare's implementation as issues are resolved by X3J16.

---

## 14.5 Suggested Reading

Booch, Grady, and Vilot, Michael. *Using Namespaces*. **JOOP** (November-December, 1993), page 14.

Koenig, Andrew. *Wrapping Up the Standard*. **JOOP** (November-December, 1993), page 67.

Lajoie, Josée. *Namespaces*. **JOOP** (November-December, 1993), page 52.



# 15

## High C++ Exception Handling

---

This chapter describes the High C++ exception-handling facilities.

---

### 15.1 Detecting Errors

When a function `f()` detects an error, a typical coding practice is to have `f()` either return a null value, assuming such a value exists, or set a global value (such as `errno`) indicating a problem. Then any client of `f()` must check the return value or global value to see if an error occurred. Furthermore, the client might have to transmit the error code to the function that called it, which then transmits the error code to the function that called it, and so forth. Every function in the calling chain must inspect an error code or global value until some function `g()` is reached that can deal with the error situation and dismiss it.

Such programming practice is complicated and prone to error.

---

### 15.2 How Exception Handling Works

*Throwing and catching exceptions* C++ exceptions are designed to handle errors in such a way that `f()` can throw an exception when an error occurs, and `g()` can catch the exception and deal with the error. None of the functions in the calling chain between `f()` and `g()` need to do anything about the error situation; the communication is directly between the source of the error and the function that deals with it.

---

#### 15.2.1 The `throw`, `try`, and `catch` Statements

*On error, a function throws an exception* When a function detects an error, it can *throw* an exception. In C++, the code for this is as follows:

```
throw expression
```

This `throw` should occur within a block of code (called a *try block*) constructed to test and catch the exception.

The value of *expression* is communicated to the *catch* point, so you can place information in the expression that tells the catch point what happened. `throw expression` is an expression of type `void`, and has the same precedence as the assignment operator `=`. For example:

```
struct File_error {
    const char *file_name;
    File_error(const char* name) {
        file_name = name;
    }
};

...
void open(const char *name) {
    if (attempt_to_open_name_fails)
        // Communicate name to catch point by construct-
        // ing and throwing a File_error object contain-
        // ing name
        throw File_error(name);
}
```

*Try blocks and handlers* To catch an exception, you must embed code that can potentially cause the exception in a try block. The try block is followed by a sequence of one or more *handlers* that specify what kinds of exceptions can be caught, catch them, and perform whatever actions are needed to deal with the error:

```
try {
    // Code that can throw an exception or calls code
    // that can do so
    open(some_file);
}
catch (File_error &F) {
    printf("File error in file %s\n",
        F.file_name);        // Catch File_errors
}
catch (Floating_overflow &O) {
    // Catch floating-point overflows
}
```

*Handlers catch exceptions* Formally, a try block has this syntax:

```
try compound-statement
```

followed by one or more handlers, each of this syntax:

```
catch (exception-declaration) compound-statement
```

Each handler takes a single argument, because the *exception-declaration* is just like an *argument-declaration*, except that you cannot specify a storage class or a default value.

The type of a handler's exception declaration specifies what sort of exception it can catch. Basically, it can catch anything thrown that is of the same type; or if the handler specifies a base type or pointer to a base type, the thrown type can be a class or a pointer to a class derived from the base. This is just the standard C++ conversion from derived to base. See §15.8: *What a Handler Can Catch* on page 236 for the complete list of rules.

---

## 15.2.2 What Happens When an Exception Is Thrown

When an exception is thrown, the exception-handling (EH) run time looks at the handlers in the order they are listed for each try block that has been entered but not exited, from most recently entered to first entered. Control is transferred to the first handler whose type allows it to catch the exception. The search continues until such a handler is found, to the end of the program if necessary. Once an appropriate handler is found, the search stops; for each exception thrown, there can be no more than one catch. When control enters the handler, the exception becomes active and remains active until handling is complete and control exits the handler. For example:

```
void throw_something() { throw 1; };    // 1:
void func3() {
    try {                                // 2:
        throw_something();
    }
    catch(char *p) { /* ... */ }         // 2a:
    catch(int *p) { /* ... */ }         // 2b:
}
void func2() {
    try {                                // 3:
        func3();
    }
```

```

        catch(float f) { /* ... */ }           // 3a:
        catch(int i)  { /* ... */ }           // 3b:
    }

    void func1() {
        try {                                   // 4:
            func2();
        }
        catch(double d) { /* ... */ }         // 4a:
    }
    main () { func1(); }

```

In this example, when **1** is thrown at **1:**, the EH run time has available to it the handlers in the three active try blocks at **2:**, **3:**, and **4:** in that order. It checks the handlers at **2a:**, **2b:**, **3a:**, and **3b:**. Handler **4a:** is not considered; searching stops at **3b:** because **3b:** catches the integer thrown.

**throw expression** initializes a temporary object of the static type of the expression (**int** in the preceding example). This copy is used to initialize the appropriately typed handler parameter; the mechanism is similar to function-argument passing. The initialization can require another copy, called the *parameter copy*, if the handler parameter is not the same type as that of the first copy made. The EH run time is allowed to optimize away copies if doing so does not change the program's behavior (except for using copy constructors to copy the object).

After a handler completes, program control goes to the statement following the sequence of handlers.

---

## 15.3 Unwinding: Automatic Clean-Up with Destructors

C++ automatically causes destructors to be called to perform clean-up on automatic objects; you do not need to write specific handlers for this purpose. This clean-up of objects is called *unwinding*.

As control passes from a **throw** to a handler, destructors are invoked for all automatic objects constructed since the handler's **try** block was entered. Any object or array of objects that is partially constructed has destructors executed only for its fully constructed subobjects. So, if a constructor for an object allocates a resource, the destructor can free the resource as exception handling *passes through* the site of the object's declaration:

```
void f() {  
    try {  
        C x;  
        throw 77;  
    }  
    catch(int) { ... }  
}
```

In this example, the destructor (if any) for object `x` is run before control enters the handler.

A constructor for an object is considered to have fully executed when all user-written code in the constructor has run. A destructor for an object is considered to have run as soon as control enters the destructor.

The definition for the constructor makes clear sense, but the destructor requires a bit more analysis. If you are in the middle of user-written code in a destructor, and an exception is thrown, you should not re-enter that same destructor for the same object during an unwind; half the destructor code has executed, so the object is in an indeterminate state. Hence the definition: as soon as user code in a destructor starts to execute, the object is considered completely destroyed for purposes of unwinding.

---

### 15.3.1 Additional Clean-Up During Unwinding

Although the functions between the `try` and the `catch` do not need to do anything about the error situation, you might want a particular function to do clean-up if an exception *passes through* it. One way to do this is to write, in that function, a try block that catches any exception, code to perform the clean-up, and a `throw` command to *rethrow* the exception once the clean-up is finished. This way, you can perform your own clean-up in addition to the automatic clean-up provided by the unwinding process. For example, in the program below, the `malloc`-ed space is not reclaimed after the throw:

```
void computation() {  
    if(some_error)  
        throw 1;  
}
```

```

void func2() {
    void *p = malloc(a_lot);
    // Does something with p
    computation();
}

void func1() {
    try { func2(); }
    catch(int i) { /* ... */ }
}

```

The thrown integer is caught in `func1()`, and the pointer allocated in `func2()` is not deallocated.

You can manually deallocate the pointer by writing a catch clause that catches any exception, performs your own clean-up, and then rethrows the exception to be caught by the handler for which it was originally intended. In the following example, `func2()` does this:

```

void func2() {
    void *p = malloc(a_lot);
    try {
        // Does something with p
        computation();
    }
    catch(...) {
        // Catches anything
        // Clean-up code, such as freeing storage
        free(p);           // Frees memory in the pointer
        throw;             // Rethrows current exception
    }
}

```

Here, the handler in `func2()` catches the integer thrown in `computation()`, does its own clean-up, freeing the allocated pointer, and then rethrows the integer, which is caught by the original intended handler located in `func1()`.

In general, `throw` without an expression rethrows the currently active exception. The *exception-declaration* “...” can catch anything. Thus, no matter what is thrown, the combination `catch(...)` and `throw;` catches the exception, passes control to whatever clean-up code you specify, and then rethrows the exception to be caught by some other handler. If a parameter copy was made, a rethrow causes destruction of the copy. The parameter of

any subsequent handler is initialized from the original copy, or from a parameter copy generated to match that parameter's type.

Because the EH makes copies, you cannot count on changes you make to a handler argument being reflected in a rethrown value caught by a subsequent handler.

---

## 15.3.2 Stacked and Nested Exceptions

*An **active** exception is one for which handling is not complete. The **current** exception is the one being handled.*

Active exceptions conform to a stack discipline. When a handler is entered, the current exception, if any, is stacked and the exception caught by the handler becomes the current one. When a handler finishes, the exception on top of the stack, if any, is popped and becomes the current one once again.

Therefore you can dynamically nest exception handling, including rethrows:

```
void f() {
    try {
        throw 77;           // 1: Throws int exception
    }
    catch(int i) {
        try {
            throw "string"; // 2: Throws string exception
        }
        catch(char *p) {
            ...;             // 3: Catches string
                             exception
        }
        throw;              // 4: Rethrows int exception
    }
}
```

Here, the **throw** at 4: rethrows the **int** value 77, which is popped off the stack and becomes the current exception as soon as control leaves the handler at 3:. If you had a **throw** at 3:, it would have rethrown the string value **string**.

If you rethrow when there is no active handler, the EH run time reports an error.

## 15.4 Exceptions During Unwinding: the `terminate()` Function

Before control enters a handler `H`, destructors called during the unwind process can throw exceptions. This is valid as long as the exception is also caught before `H` is entered. If an exception is thrown during destruction that attempts to enter or skip past `H`, the run-time function `terminate()` is called:

```
struct C { ~C() { throw "skip"; } };    // 3:
void f() {
    try {
        try {
            C x;
            throw 1;                      // 1:
        }
        catch(int i) {                   // 2:
        }
    }
    catch(const char *) {                // 4:
    }
}
```

Here, `1:` throws an integer, and in searching for a handler, the EH run time finds `2:`. Before transferring to `2:`, object `x` is destroyed. However, at `3:` the destructor throws a string. Rather than have the string caught at `4:`, the EH run time detects an attempt to skip over `2:`, and calls `terminate()`.

`terminate()` is also called if no handler is found for a thrown exception or when the exception mechanism finds the stack corrupted.

*`terminate()` calls `abort()` unless you specify a different function*

The default behavior of `terminate()` is to call `abort()`. You can change the function that `terminate()` calls by calling `set_terminate()` with a function pointer; then `terminate()` calls the function pointed to instead of `abort()`.

`set_terminate()` returns a pointer to the function `terminate()` would have called, and records your function as the one to call. In this way, you can program a chain of functions to be called when `terminate()` is invoked. It is an error if the function you supply returns to `terminate()` or throws an exception.



## 15.5 Partially Constructed and Destroyed Heap Objects

An issue not discussed in the ANSI X3J16 draft on exception handling is the treatment of partially constructed and partially destroyed heap objects. The sentiment is that partially constructed heap objects should be destroyed; therefore, if a **new** operation is not complete, the destructors for objects constructed so far are run as part of exception unwinding:

```
int cnt = 0;
struct S {
    S() { if (++cnt == 3) throw 1; }
    ~S();
};
void f() {
    try {
        S *a = new S[3];
    }
    catch(int i) {
        // ...
    }
}
```

Here the constructor for **S** throws an exception after two objects have been constructed. The third element of the array is not constructed, so the **new** operation is unfinished, and exception handling runs the destructors on **a[0]** and **a[1]** before entering the handler. It also frees the storage allocated by **new**.

A somewhat more controversial decision that remains to be resolved is what to do about partially destroyed heap objects. We expect the consensus to be “finish the destruction”, which is what MetaWare has implemented. A delete operation that has started but has not finished is completed as part of unwinding:

```
int cnt = 0;
struct S {
    S() { }
    ~S() { if (++cnt == 2) throw 1; }
};
```

```
void f() {  
    try {  
        S *p = new a[3];           // This completes  
        delete p;                  // This throws  
    }  
    // Deletion completed as part of unwinding  
    // before control enters catch(int i)  
    catch(int i) {  
        // ...  
    }  
}
```

The destructor throws an exception after `a[2]` and `a[1]` are destroyed; only `a[0]` remains. The EH run time calls the destructor to destroy `a[0]` before entering the handler.

---

**Note:** There is an apparent inconsistency in the treatment of heap-based objects. Completely constructed heap objects are *not* automatically destroyed as the EH run time unwinds, but partially constructed and partially destroyed objects are, because it is difficult for a user to program the destruction of objects that are in a partially constructed state.

Objects constructed via a completed `new` operation are not destroyed because it is impossible for the EH run time to know what completely constructed heap objects should be destroyed; the pointers to such objects could have been copied anywhere.

---

## 15.6 Function Exception Specifications

In a function declaration, you can list the set of exceptions that a function might directly or indirectly throw. The list is known as an *exception specification*, and takes this form:

```
throw ( argument_decl_list )
```

The exception specification can appear in the same place as a function declaration's optional *cv\_qualifier\_list*.

*Function* The list of argument types in an exception specification must have no default values and cannot use "...". The exception specification is checked at run time only after a handler has been found for a thrown exception. If the function directly or indirectly throws an exception that could not be caught by a handler of the form `catch(T)`, where `T` is one of the exception-specification types, a function named `unexpected()` is called. For example:

```
void f() throw(A*, int) {
    // ...
}
```

is effectively the same as:

```
void f() {
    try {
        // ...
    }
    catch (A*) { throw; }
    catch (int) { throw; }
    catch (...) { unexpected(); }
}
```

*`unexpected()` calls `terminate()` unless you specify a different function* The default behavior of `unexpected()` is to call `terminate()`. You can program the behavior of `unexpected()` by calling `set_unexpected()` and passing it a function pointer. `set_unexpected()` returns the pointer that `unexpected()` would have called, and records your function as the one to call. In this way you can program a chain of functions to be called when `unexpected()` is invoked.

Unlike `terminate()`, `unexpected()` can throw an exception. Handlers for this exception are searched for starting at the call of the function whose exception specification was violated. Thus an exception specification does not guarantee that only the listed types will be thrown. After the call to `unexpected()`, the function behaves as if it had no exception specification.

## 15.7 Miscellaneous Rules for Handling Exceptions

The following rules should help you write effective exception-handling code:

- In a sequence of handlers, it is an error for handler *H1* to precede handler *H2*, if *H1* always catches an exception that *H2* can also catch. This erroneous situation ensures that *H2* will never run:

```

struct base { ... };
struct derived: base { ... };
try {
    // ...
}
catch(base *bp) { }
catch(derived *dp) { }    // Error: Control can never
                          // enter this handler.

```

- A throw can appear as conditional operands of `?:`; the resulting type of the `?:` operator is the type of the other conditional operand:
 

```
x ? throw something : 1    // Type is int.
```
- You cannot jump into a try block or a handler, but you can jump out of either.
- Exceptions are not handled for globally declared objects.

## 15.8 What a Handler Can Catch

Here is the full list of rules for what a **catch** clause can catch. A handler can catch a thrown object under the following circumstances:

- The thrown object can be copied at the throw point (for classes, this means that a default copy constructor is accessible; for non-classes, this requirement is irrelevant).
- A handler whose type is *T*& can catch the same things that a handler of type *T* can catch.

- A handler whose unmodified type is *T* can catch a thrown object whose unmodified type is *E* if any one of the following is true:
  - *T* and *E* are the same type.
  - *T* is an accessible base class of *E* at the throw point.
  - *T* is a pointer type and *E* is a pointer type that can be converted to *T* by a standard pointer conversion.
- A handler whose *exception-declaration* is “...” can catch anything.

These rules are slightly different from, and less limiting than, those in the current ANSI X3J16 draft. Most of the X3J16 rules will undergo change.

---

## 15.9 Suggested Reading

Christian, Kaare. *Making Exceptions with C++*. **PC Magazine** (December 21, 1993), page 311.

Lajoie, Josée. *Exception Handling: Supporting the Runtime Mechanism*. **C++ Report** (March-April, 1994), page 36.

Reed, David R. *Using C++ Exceptions*. **C++ Report** (March-April, 1994), page 18.

Swan, Tom. *Trouble in Paradise*. **Dr. Dobb's Journal** (May, 1994), page 123.



# 16

# C++ Run-Time Type Information (RTTI)

---

C++ allows implicit conversions from a derived class to a base class. That is, an object of a derived class contains a subobject of the type of one of the bases. For example:

```
struct D : B { ... };
void f(B* bp) { ... }
...
D *dp = new D;
f(dp); // Automatic conversion to pointer-to-base
```

The conversion in the opposite direction is not automatic, because a pointer to a base object (say of class **B**) is not necessarily a pointer to a larger object of derived class **D**. You might just have a **B**, not a **B** within a **D**. Therefore C++ requires casting to convert in the opposite direction. With a cast, you assert to the compiler that a pointer is in fact a pointer to a **D** object:

```
B *bp = dp;           // Points to base
D *dp2 = (D *)bp;     // Converts back; points to D object
```

This process works only if the base is not virtual. If the base is virtual, the compiler cannot generate code to make the conversion to **D\***, and the cast elicits an error message.

These problems are solved with *run-time type information* (RTTI), added to C++ in March 1993.

---

## 16.1 RTTI and **typeid**

*The complete type of an object* For subobjects of polymorphic classes (those that have virtual functions), with RTTI you can find out the *complete type* of a subobject, the ultimate derived class. For example, you can write the following code:

```
B *bp = new D;
```

Keyword **typeid** Although the *static* type of **bp** is pointer to **B**, the *complete* type of **bp** is pointer to **D**, because **bp** points to a subportion of a **D** object. You can use the keyword **typeid** to obtain a representation of type **D**:

```
typeid(*bp);
```

The representation takes the form of another object corresponding to type **D**.

Other benefits of RTTI include being able to do the following:

- print out the name of the complete type of an object
- find out the size of an instance of the complete type
- inquire as to the base classes

For example:

```
printf("bp really points to a %s", typeid(*bp).name());
```

prints:

```
bp really points to a D
```

## 16.2 The Type Object

**typeid(expression)** or **typeid(type)** returns a reference to an object of type **const Type\_info**, or of some class derived from **Type\_info** (**class Type\_info** is defined in **typeinfo.h**). This object is called a *type object*.

For **typeid(type)** (similar to **sizeof(type)**), the type object describes exactly the type given:

**Table 16.1** Objects Returned by **typeid** Expressions

The expression:	Returns a reference to a <b>Type_info</b> object:
<b>typeid(int)</b>	Describing type <b>int</b>
<b>typeid(int *)</b>	Describing pointer to <b>int</b>
<b>typeid(void (*)(int))</b>	Describing pointer to a function taking <b>int</b> and returning <b>void</b>
<b>typeid(D)</b>	Describing class <b>D</b>



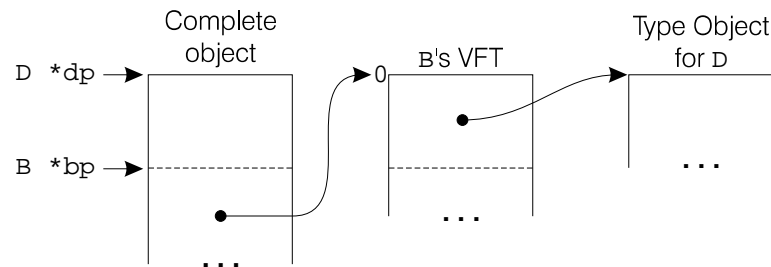
For `typeid(type)`, the type object is statically known by the compiler; you specify precisely the type. The same holds for an expression of anything other than a polymorphic class: the type is what is statically known by the compiler, as in the following examples.

```
int i;
typeid(i);           // Same as typeid(int)
int *p;
typeid(p);           // Same as typeid(int *)
B *bp;
typeid(bp);          // Same as typeid(B *)
struct s {};
struct d:s{};         // Derived from s
s *sp = new d;        // Cast from derived to base
typeid(*sp);          // Same as typeid(s), because s
                      // isn't polymorphic
```

But when the argument is an object of a polymorphic class, the compiler generates code to fetch the true complete type:

```
typeid(*bp);         // Produces typeid(D)
```

*Polymorphic* We call this a *polymorphic* `typeid`. The code fetches a pointer to the type object describing the complete type from position 0 of the polymorphic class's virtual function table. That is why the class must be polymorphic, as is shown in Figure 16.1.



**Figure 16.1** A Polymorphic `typeid`

Since its first release, MetaWare's implementation has reserved location 0 (zero) of the virtual function table for just such a purpose.

*The single-copy heuristic* Type objects take up space, and there might be duplicates of them. For example, each compilation unit containing `typeid(int)` has a distinct type object

describing the type `int`. Duplicates can be avoided for classes. A class has just one copy of its type object if it has a non-pure-virtual, non-inline member function. The type object is placed in the module containing the definition of this function (we say the class satisfies the *single-copy heuristic*).

The single-copy heuristic is also used by the compiler to determine the location of the unique copy of a class's virtual function table; but the heuristic is used for type objects whether or not the class has virtual functions. For example:

```
struct s {
    int f()           // Where s::f is, so is typeid(s)
    int g();
};
struct t { };        // Every occurrence of typeid(t)
                     // produces a distinct copy.
```

---

## 16.2.1 Using the Type Object

You can use the type object for other things than printing out the name of a type. For example, suppose you have a pointer to a base object, and you want to call a function that is defined only on the complete object:

```
((D*)bp)->dfunc();
```

*Checking for `typeid` equality* You want to make this cast secure, in case the base is not part of a `D`. One way you can do this is by checking for `typeid` equality:

```
if (typeid(*bp) == typeid(D))
    ((D*)bp)->dfunc();
```

To use `==`, include `typeinfo.h`, the header file in which `Type_info` is defined.

However, checking `typeid` equality is not the best way to make sure your base object is part of `D`, because in fact `*bp` might be part of an object that is even larger than a `D`; that is, it might be of a type derived from `D`. The previous example works if the complete type is exactly a `D`.

*dynamic\_cast* To determine if `*bp` belongs to `D` or to a class derived from `D`, you use `dynamic_cast` (see §17.3 on page 249):

```
D *try_dp = dynamic_cast<D *>(bp);
```

`dynamic_cast` fetches the type object from `*bp`'s virtual-function table, then does the following:

- If `*bp`'s complete type is `D`, `dynamic_cast` simply makes sure that `B` is an unambiguous base of `D`, and then does the conversion from the base to `D`.
- If `*bp`'s complete type is `E`, derived from `D`, `dynamic_cast` converts `B` to `E` unambiguously, then converts `E` back to base `D`.

In both cases, the pointer returned is 0 (zero) if `*bp` is not part of a `D`, or the pointer to the `D` object if `*bp` is part of a `D` or class derived from `D`.

This means you can write the following:

```
D *try_dp = dynamic_cast<D *>(bp);
if (try_dp != 0)
    try_dp->dfunc();
else
    printf("Sorry, bp doesn't point to a "
           "piece of a D.");
```

`dynamic_cast` is described in detail in §17.3: `dynamic_cast<Type>(expr)`.

---

## 16.2.2 Explicit Initialization in Conditions

With the introduction of RTTI, C++ was also extended to allow declarations with explicit “=” initialization in conditions. For example:

```
if (D *try_dp = dynamic_cast<D *>(bp))
    try_dp->dfunc();
else
    printf("Sorry, bp doesn't point to a "
           "piece of a D.");
```

*Syntactic  
ambiguity:  
declarations  
versus casts*

Unfortunately, this added syntax conflicts with previously correct programs. However, conflicts are expected to be rare:

```
if (Type(X) = 3) ...
```

`Type(X) = 3` could be a declaration of `X` of type `Type`, initialized to 3. Or it could be the type cast of `X` to `Type` and a subsequent assignment to the result of the cast.

Because of the ambiguity, High C++ also provides syntax where the declaration can be followed by a “;” to specify a declaration unambiguously:

```
if (Type(X) = 3;) // Declaration, no question
                // about that!
```

To sum up:

- Use `dynamic_cast` when you want to know if an object is part of a bigger type `D`, but you do not care whether the complete type is derived from `D`.
- Use equality of `typeid`s when you want to know if an object is exactly of a certain type.

## 16.3 Functions on Type Objects

The following functions, available on all type objects, are declared in `typeid.h`:

```
const char* name() const;    // Simple name
const char* full_name() const;
                           // Full name: w/mangling
int Type_info::operator ==(const Type_info&) const;
int Type_info::operator !=(const Type_info&) const;
int before(const Type_info&) const;
long size() const; // Size of an object of this type
```

These functions on type objects do the following:

<code>name()</code>	Returns the name of the type, suitable for printing.
<code>full_name()</code>	Returns the fully mangled name of the type. Whereas <code>name()</code> might be the same for two distinct types (say, a global class <code>X</code> and a class <code>X</code> nested within another class <code>Y</code> ), <code>full_name()</code> is guaranteed to be distinct for distinct types.
<code>==</code> and <code>!=</code>	Compare for equality, by comparing the two <code>full_name()</code> functions; the comparison is not cheap, although a hash code makes the string compare unnecessary for most non-equal types.

### Extension

**Extension**

<code>before()</code>	Sorts type objects; you can use it for implementing an efficient associative array indexed by type objects. <code>before()</code> lexicographically compares the <code>full_name()</code> functions of the two types to determine the order.
<code>size()</code>	Returns the total number of bytes required to store an instance of the type.

---

**Note:** `size()` and `full_name()` are MetaWare extensions.

---

Suppose you write a routine that takes an arbitrary `Type_info` object and prints its name:

```
void f(const Type_info& tid) {
    printf("type:%s ",tid.name());
}

...
int a, *b;
f(typeid(b));
f(typeid(a));
```

Suppose you want to print more information if the type object turns out to be of a particular kind of type. For example, you might want to print the pointed-to type of a pointer type. `typeinfo.h` contains the definition of `Type_info` and of several derived classes. The derived classes are a MetaWare extension, and they classify the kind of types a type object can represent. The derived classes are as follows:

```
class Modified_type_info: public Type_info
class Pointer_type_info: public Type_info
class Member_pointer_type_info: public Type_info
class Array_type_info: public Type_info
class Func_type_info: public Type_info
class Class_type_info: public Type_info
```

---

**Note:** These classes answer the need of the exception-handling implementation for dynamic type information.

---

If a `Type_info` is actually a `Pointer_type_info`, the type is a pointer type, and you can discover the type it points to. You first determine that a `Type_info` is a pointer type by using `dynamic_cast` to determine the complete type of a particular `Type_info` (because `Type_info` is a polymorphic class, you can inquire as to the true type of the `Type_info` object):

```
void f(const Type_info& tid) {
    printf("type:%s ",tid.name());
    if (Pointer_type_info *PTI =
        dynamic_cast<Pointer_type_info*>(&tid))
        printf("pointed-to type is %s",
            PTI->base_type->name());
}
```

---

**Note:** Be sure to use `dynamic_cast` with pointer types, not reference types, because `dynamic_cast` throws an exception if a reference cast fails.

---

You can also find out whether the `typeid` of the `Type_info` is that of `Pointer_type_info`, and use an explicit cast:

```
void f(const Type_info& tid) {
    printf("type:%s ",tid.name());
    if (typeid(tid) == typeid(Pointer_type_info)) {
        Pointer_type_info *PTI =
            (Pointer_type_info*)(&tid);
        printf("pointed-to type is %s",
            PTI->base_type->name());
    }
}
```

This is generally much more efficient than using `dynamic_cast`.

# New Cast Notation for C++

---

Four new forms of cast notation to express the safe casting of one type to another were added to the C++ language at the San Jose meeting of X3J16 in November 1993. These are the four forms of cast:

```
dynamic_cast<Type>
static_cast<Type>
reinterpret_cast<Type>
const_cast<Type>
```

In the MetaWare implementation, these are four predefined templates:

```
template <class Type> class dynamic_cast { ... };
template <class Type> class static_cast { ... };
template <class Type> class reinterpret_cast { ... };
template <class Type> class const_cast { ... };
```

Types that are instantiations of any of these four templates behave in a special manner when they are used in a cast. `Q<Type>(expr)`, where `Q` is one of the four new MetaWare templates, actually casts `expr` to type `Type`, subject to conditions described in the following sections. For example:

```
static_cast<Type>(expr)
```

casts `expr` to `Type`. So does this:

```
typedef static_cast<Type> sc_Type;
... = sc_Type(expr);
```

Each of the last three templates expresses approximately a subset of the full potential of a C++ cast. `dynamic_cast<Type>` gives the ability to safely cast from, say, a pointer-to-base to a pointer-to-derived, even when the base is virtual. This ability has not previously been present in C++.

---

## 17.1 `static_cast<Type>(expr)`

This is a reasonably safe subset of `(Type)(expr)`. The result is of type `Type`.

If *expr* is of type *Te*, the cast of *expr* to type *Type* is valid if it falls into any one of the following cases:

- *Te* => *Type* is a standard conversion.
- *Type* => *Te* is a standard pointer, reference, or pointer-to-member conversion (this covers contravariant conversions); *Te* is not a virtual base of *Type*, and *Te* => *Type* does not cast away **const** (see §17.2).

**Note:** A *contravariant conversion* is a conversion from pointer-to-base to pointer-to-derived, or from reference-to-base to reference-to-derived.

- *Type* is **void** or a class type, and (*Type*)(*expr*) is valid.

Semantics of the conversion are the same as for (*Type*)(*expr*).

Thus, conversions from pointer-to-non-virtual-base to pointer-to-derived are permitted, but dangerous conversions from type **int** \* to **char** \* or from pointers to two unrelated class lattices are not.

To safely convert a virtual base to a derived class, use `dynamic_cast`; to convert dangerously, use `reinterpret_cast`.

---

## 17.2 const\_cast<Type>(expr) and Casting Away const

*Definition of casting away const* Casting away **const** means converting a pointer-to-**const** to a pointer-to-non-**const**. To cast away **const**, use `const_cast`.

Casting away **const** also incorporates casting away **volatile**, and handles multi-level pointer conversion. It is defined as follows: The conversion

*Type* cv \* => *Type'* cv' \*

where *cv* is one of the four sets:

{**const**}, {**volatile**}, {**const**, **volatile**}, {}

is said to cast away **const** if one of the following is true:

- *cv* is not a subset of *cv'*.
- *Type* and *Type'* are pointer types, and *Type* => *Type'* casts away **const**.



For example:

```
const int * => int *
```

and

```
const int * * * * => int * * * *
```

both cast away `const`. However:

```
const int * * => int *
```

does not cast away `const`, because there is an implicit conversion of `const int *` to `int`, which is not subject to the rules for casting away `const`. This last case is irrelevant to `static_cast` because you can never have an unequal number of `*`s; however, it is relevant to `reinterpret_cast`, which also cannot cast away `const`.

---

## 17.3 `dynamic_cast<Type>(expr)`

`dynamic_cast<Type>(expr)` allows safe conversion from, for example, a pointer-to-base to a pointer-to-derived. Run-time checks are used to be sure the conversion is completely valid. To do the run-time checks, run-time type information must be available for both the source and destination types, or the cast might fail. A failure of the cast either returns 0 (zero) or throws an exception, depending on the nature of the failure.

The result of the cast is type `Type`. `Type` must be one of the following:

- a pointer to a fully defined class
- a reference to a fully defined class
- `void *`

Let `Te` be the type of `expr`. Then, if the following are *all* true:

- `Type` is a pointer to a base class `B`.
- `Te` is a pointer to a derived class `D`.
- `B` is an accessible base class of `D`.

the result is the same as `(D*)B`. The same is true if `Type` and `Te` are reference types that are similarly related.

Otherwise, `Te` must be a polymorphic type. That is, it must have virtual functions, so that `expr`'s true run-time type can be accessed from the virtual function table.

If `Type = void *`, the result is the pointer to the complete object pointed to by `expr`.

Otherwise, a run-time check is applied to see if `expr` can be converted to type `Type`. Basically, `expr` is first converted into a pointer to the complete object; then this pointer is converted to `Type`. The first conversion fails if `expr` does not point to an unambiguous base class of its complete type. If either conversion cannot be done, the result is 0 (zero). If `Type` is a reference type, and the cast fails, exception `Bad_cast` is thrown. (See Chapter 15: *High C++ Exception Handling* on page 225 for general information on exception handling.)

---

## 17.4 `reinterpret_cast<Type>(expr)`

A `reinterpret_cast` allows “raw” conversion from:

- any pointer type to any other pointer type that is big enough
- any pointer type to any integer type that is big enough
- any integer type to any pointer type
- any object `E` to any reference type `Type`, provided that `reinterpret_cast` can convert `&E` to `Type*` by a conversion in one of the preceding three bullets.

No class hierarchy navigation is done; basically, when the type sizes are the same, the cast value is unchanged. When the sizes of the types are different, the compiler warns. For conversion to a larger type, the semantics are equivalent to `*((Type*)&expr)`. This implies that the result contains some indeterminate value.

# A

## Migrating from C to C++

---

The High C/C++ compiler supports the gradual introduction of C++ features into C programs. You can use almost all of C++ — intermixing it with C — while preserving the semantics of your C code.

*Incremental C++* C++ is almost, but not quite, a superset of C. A C++ compiler must reject some C programs. But with High C/C++, you can introduce C++ constructs in your C program without affecting whether your C program text will compile, and without affecting the result when it does compile. We call this the *Incremental C++* approach.

---

### A.1 C/C++ Compatibility Issues

The **Annotated C++ Reference Manual** discusses C/C++ compatibility issues, and the ANSI C++ standardization committee (of which MetaWare is an active member) is creating a (much) longer list of issues. We describe a few of the issues here to help you decide if you want to use Incremental C++.

**C ≠ C++**

The following examples present C code that works with a C compiler, but either fails to compile or works differently with a C++ compiler. All the examples compile and work correctly with MetaWare's Incremental C++.

#### **Example A.1** *structs Are Scopes in C++*

---

```
struct s {  
    enum color {red,white,blue} x;  
};  
// color and red are known here in C.  
enum color z = red;
```

This program fails to compile in C++ because `color`, `red`, `white`, and `blue` are local to the `struct`. Instead, in C++ you can write the following:

```
enum s::color z = s::red;
```

---

**Example A.2 C++ Has Additional Keywords**

---

```
type_handle make_primitive_type(type_class class,  
                                int len);
```

---

This excerpt from compiler source code fails to compile in C++ because the reserved word **class** is used as a parameter name.

---

**Example A.3 sizeof char Is Different in C++**

---

```
if (sizeof('a') != sizeof(int))  
    printf("different from C!\n");
```

---

This fragment prints nothing when compiled as C. It prints a message when compiled as C++, because the **sizeof** a character in C++ is typically one byte: not the same as **sizeof(int)**, as in C.

---

**Example A.4 A Tag Is a Type Name in C++**

---

```
typedef int * T;  
struct T { int x; };
```

---

This is an error in C++ because **T** is a type name for **int \*** and for the **struct**, so you have a duplicate declaration of **T**.

---

## A.2 Using Incremental C++

What C++ features can you use with Incremental C++? You can use everything except these:

- templates
- class names (tags) redefined as something other than class names
- **typedefs** within classes

You can use constructors, destructors, overloaded functions, member functions, virtual functions, derived classes, virtual bases, and so forth. Some features require modified syntax, described in the rest of this appendix.

*Libraries* The MetaWare High C++ include files are designed for full C++, not Incremental C++, so you cannot use those files incrementally. If you need the features of the C++ library, access them through separate modules you write in full C++.

*Command-line option for Incremental C++* How do you use Incremental C++? An Incremental C++ program is a C program — and so has the `.c` suffix — but you tell the compiler to enable Incremental C++ features with a special option that indicates the level of C++ compilation you want. See the **High C/C++ Programmer's Guide** for information about specifying the C++ level.

---

## A.2.1 Using Special Keywords and Functions

You can access many Incremental C++ features by using keywords that do not intrude on the namespace of C programs, and by using slightly different syntax in some places. See §A.4: *Moving from Incremental C++ to Full C++* on page 257 for more information. The additional keywords are shown in Table A.1.

**Table A.1** *Incremental C++ Keywords*

Use This Incremental C++ Keyword	Instead of This C++ Keyword
<code>_Class</code>	<code>class</code>
<code>_Virtual</code>	<code>virtual</code>
<code>_Operator</code>	<code>operator</code>
<code>_Private</code>	<code>private</code>
<code>_Protected</code>	<code>protected</code>
<code>_Public</code>	<code>public</code>
<code>_Friend</code>	<code>friend</code>
<code>_Delete</code>	<code>delete</code>
<code>_New</code>	<code>new</code>

**Table A.1 Incremental C++ Keywords (Continued)**

Use This Incremental C++ Keyword	Instead of This C++ Keyword
<code>_Inline</code>	<code>inline</code>
<code>_Anonymous</code>	none

For example, you might write the Incremental C++ program:

```
_Class C {
    _Virtual ~C() { _Delete p; }
    _Friend struct s;
    C *p;
};

struct s { int x; };    // ANSI C here
```

and compile this as a `.c` file with `-Hcpp1v1=1` (see the **High C/C++ Programmer's Guide** for information about this command-line option).

Other C++ features that do not require keywords are automatically available in Incremental C++, such as the initialization of static variables with non-constant expressions. For example:

```
struct s {int a,b; } X = {3,4};
int k = X.a + X.b;    // Generates initialization code
```

## A.2.2 Accessing C++ Features Incrementally

*Alternate keywords* Use alternate keywords. Use `_Virtual`, `_Operator`, `_Private`, `_Protected`, `_Public`, `_Friend`, `_Delete`, and `_New` instead of the normal C++ keywords.

*Anonymous unions* Use `_Anonymous` when you want an anonymous union. (In ANSI Standard C, there is no such thing.) For example:

```
_Anonymous union { int a,b; };
int c = a+b;    // a and b are visible
```

Within a `_Class`, you can use `_Anonymous`, though it is not required; anonymous unions are accepted. With a `struct` or `union`, you must use `_Anonymous` if that is what you want:

```

    _Class C {
        union {int a,b;};    // OK, anonymous union
    };

    struct S {
        _Anonymous union {int a,b; };
        // _Anonymous union
        // Without _Anonymous, this would be a
        // vacuous declaration, as in ANSI Standard C.
    };

```

*Overloadable functions* Make functions overloadable. Functions not contained within a **\_Class** are not overloadable by default. To make them overloadable, surround such functions by **extern "C++" { ... }**. For example:

```

extern "C++" {
    void f() {...}
    void f(int) {...}
};
extern "C++" void f(float) {...};

```

This example declares three separate functions named **f**.

You must surround all declarations and definitions of a function **f** with **extern "C++"** for **f** to be overloadable. That is, you cannot have one declaration that is not overloadable and subsequent declarations that are.

You might consider surrounding your whole program with **extern "C++" {...}**.

Functions contained within a **\_Class** are overloadable. You do not need to surround the **\_Class** declaration with **extern "C++" {...}**.

*Avoiding ambiguities with “..”* Use special syntax. You must employ the syntactic element “..” in several contexts to use certain C++ constructs that introduce parsing ambiguities or difficulties in the language. The “..” prevents these ambiguities and allows the programs to be parsed as Incremental C++ programs.

These are the contexts:

- Declaring a **const** or **volatile** function:

```

    _Class S {
        void f() .. const;
    };
    void S::f() .. const { /* body */ }

```

- Using a function-style cast:

```
_Class s { s(); s(int); };
s fnc(int i) { return i ? s..() : s..(3); }
```

- Using the C++ parenthesized initializer:

```
_Class s { s(int); };
main () {
    s x..(3);
}
```

- Supplying arguments to **new** after the type name:

```
_Class s { s(int); };
main () {
    new s..(3);
}
```

**void** return type for  
constructors

Specify a **void** return type. To avoid a parsing problem introduced by C++, you must specify a **void** return type for constructors and for the definition of destructors. You could use a name such as **Tor**, and **#define** it to be **void**:

```
#define Tor void
_Class s {
    Tor s();
    ~s();           // Tor is not needed here.
};
Tor s::s() { ... }
Tor s::~s() { ... }
```

This is necessary only when the constructor name is a **typedef** name.

**structs** and **unions** do not introduce **typedef** names, so you would not need **Tor** to specify their constructors:

```
struct s {
    s();           // OK
    ~s();          // OK
};
s::s() { ... }
s::~s() { ... }
```



## A.3 C++ Features You Cannot Use in Incremental Mode

What can you *not* do in Incremental C++?

No **typedef** names in structures You cannot use **typedef** names within structures:

```
_Class C {
    _Class D { };
    D *p;           // This works.
    typedef enum {red,white} color;
    color *c;       // This does not work.
};
```

Instead, move the **typedef** out of the structure.

Avoid or redefine nested class names **\_Class** names are introduced as **typedefs**, but their scope extends just as a C structure tag does. That is, the class name is not restricted to a class scope in which it might appear. If you avoid nested classes this is not a problem. If you use nested classes, the only problem is redefining the nested class name at the global scope:

```
_Class C {
    _Class D { };    // D is a typedef name.
};
// D is still declared here, unlike full C++
int D;             // This is an error in
                   // Incremental C++.
```

Use **typedef** for **struct**, **union**, and **enum** tags do not become **typedefs** as they do in C++. If you want them to be **typedefs**, use **typedef**, as is common practice in C:

```
struct s { /* ... */ };
s *p;      // Invalid in C and Incremental C++
typedef struct t { /* ... */ } t;
t *p;      // Valid in C and Incremental C++
```

## A.4 Moving from Incremental C++ to Full C++

Using macros to simplify migration Minor syntactic changes are required to convert an Incremental C++ program to a full C++ program. You can make it easy to change the syntax by using macros in your Incremental C++ programs. We recommend you use a set of

**#defines**, such as the following, placing them in your `hc.pro` profile, or including them in a suitable header file.

```
#if !__cplusplus
// Macros for Incremental C++:
#define Tor          void
#define class        _Class
#define virtual      _Virtual
#define operator     _Operator
#define private      _Private
#define protected    _Protected
#define public       _Public
#define friend       _Friend
#define delete       _Delete
#define new          _New
#define anonymous     _Anonymous
#define inline       _Inline
#define is           ..
#define init         ..
#else
#define Tor
#define is
#define init
#endif
```

Using macros `is` and `init` in place of `..` makes semantic sense, and they can be **#defined** away for full C++:

```
_Class s {
    void f() is const;
    s();
    s(int);
};

void s::f() is const { /* body */ }
s fnc(int i) { return i ? s init() : s init(3); }
s x init(3);
void *p = new s init(3);
```

# B

## High C/C++ Extensions

---

This [appendix](#) describes several language extensions that can improve the readability and augment the functionality of your programs. To maintain code portability, High C/C++ provides a macro that allows you to exclude the extensions when necessary (see §B.12: *Predefined Macro `__HIGHC__`* on page 272).

---

### B.1 Underscores in Numbers

Numbers — both floating-point and integer constants — can be written with the character “`_`” among the digits. Generally, “`_`” takes the place of the English comma in numbers. For example:

```
1_000_000           // One million
```

---

### B.2 Significant Characters in Identifiers

All characters in identifiers are significant.

---

### B.3 Case Ranges in `switch` Statements

High C/C++ allows case ranges in the `switch` statement. A *case range* is the extension:

```
case E1..E2:
```

where the meaning is equivalent to:

```
case E1: case E1+1: case E1+2: ... case E2:
```

For example:

```
switch (Ch) {
    case 'A'..'Z':    Scan_id();
                    break;
    case '0'..'9':    Scan_number();
                    break;
    default:          Scan_delimiter();
                    break;
}                  // The last break is for safety,
                  // in case the cases get reordered.
```

See also §B.15.1: *ANSI Preprocessor Numbers* on page 276.

---

## B.4 Initializing Automatic Aggregates

High C/C++ permits automatic aggregates to be initialized with arbitrary non-constant expressions, where ANSI Standard C restricts initialization lists to constant expressions. The following is valid in High C/C++, but is *not* permitted in ANSI Standard C:

```
void fnc (int i) {
    int x[5] = { 0, i, 2*i, 4*i, 8*i};
    ...
}
```

---

## B.5 Reserved Word **pragma**

In High C/C++, **pragma** is a reserved word. You cannot use the word **pragma** as an identifier. Pragas are used to direct the compiler when more than one possibility is provided for an implementation-defined aspect of compilation. Subsequent to High C/C++'s choice of **pragma**, the ANSI C committee chose **#pragma** for the same purpose.

High C/C++ supports **#pragma**, and continues to support **pragma** for compatibility. We suggest using **#pragma** for portability. However, ANSI's **#pragma** cannot appear within macros, whereas High C/C++'s **pragma** can, so there is a definite advantage to **pragma**.

---

## B.6 Named Parameter Association

Functions declared with parameter names can be called with the named parameter association syntax of Ada. Such calls refer to the parameter *names* rather than their *positions* in the argument list, so that the order of actual parameters is irrelevant.

The syntax is like that of a normal function call, except that each actual parameter expression is preceded by the corresponding formal parameter name followed by “=>”. For example:

```
typedef enum {Red, Green, Blue} Color;
void P(int A, float B, Color C, Color D) { ... } ...
P(C => Red, D => Blue, B => x*10.0, A => y);
```

You can also start the function call using positional parameter notation and then switch to named association as you add parameters from left to right. However, you cannot switch back to positional notation for later parameters; nor is any other variation allowed. Here is an example of valid parameter notation:

```
void Plot(Xlo, Xhi, Ylo, Yhi, Xinc, Yinc)
    float Xlo, Xhi, Ylo, Yhi, Xinc, Yinc; {
    ...
}
...
Plot(Alo,Ahi,Blo*2.0,Bhi*2.0,Yinc=>y,Xinc=>f(x+z));
```

---

### B.6.1 Named Parameter Syntax

The syntax for arguments to a function call is expanded from:

```
zero or more
    assignment expression
separated by commas
```

which is the normal “positional” form for ANSI Standard C, to also include:

```
zero or more
    assignment expression followed by a comma,
```

followed by  
 one or more  
*identifier* followed by => and an  
*assignment expression*  
 separated by commas.

The effect is that there can be an initial portion of the argument list in the positional form (a “positional prefix”), followed by named-parameter association form (“named parameter association suffix”).

Thus:

```
f(1, 2, 3, aa => x+y, bb => q);
```

is allowed, but:

```
f(1, 2, 3, aa => x+y, bb => q, 7, 8, 9);
```

is not.

---

## B.6.2 Constraints on Using Named Parameters

Any function called with named parameter association must specify the names of all its arguments. For example, `int f(float g);` can be called this way, but `int f(float);` or `int f();` cannot.

The identifiers in a named parameter association form of call must collectively be the names of those parameters to the called function whose values are not supplied positionally. Values for all parameters must be supplied through either the positional prefix or through the named parameter association suffix.

Given these constraints, you can transform the function call into a purely positional form. Further constraints and semantics are then as if the arguments to the call were thus transformed. For example, the call to `P()` in the first example in this section can be transformed from:

```
P(C => Red, D => Blue, B => x*10, 0, A => y);
```

to:

```
P(y, x*10.0, Red, Blue);
```

and then the regular constraints and semantics of ANSI Standard C must be obeyed for the transformed call.

---

## B.7 Nested Functions and Full-Function Variables

In High C/C++, functions can be defined within functions. Such functions are called *nested*. This facility endows High C/C++ with an expressive power found in Pascal.

---

### B.7.1 Up-Level Referencing

In the body of a nested function *N*, any name in a containing scope can be used. That is, the body of *N* can use names local to other functions that contain *N*. This is called *up-level referencing* and any such names are said to be *up-level referenced* from *N*. The single restriction is that **register**-class variables cannot be up-level referenced.

*Locals of enclosing functions*

You can achieve up-level referencing by making available to *N*, at each call to it, a way to reference the collection of locals of each of its enclosing functions. These collections are together called *N*'s *environment*. The function *P* immediately enclosing *N* is called *N*'s *parent*; the next enclosing function *G* its *grandparent*; and so on. The collection of locals of each function is called its *frame*.

*Referencing the enclosing function's frame via static link*

*N*'s environment can be implemented by passing to *N*, at each call to it, a “hidden” parameter that is a reference to *P*'s frame. If this is done for all functions, *P* will have in its frame a similar hidden parameter that links it to *G*, and so on out to the global level where functions do not need such a link. This is called the *static link* method of implementing up-level referencing, the method of choice for best efficiency and optimization.

*You cannot take the address of a nested function*

A *major difference* between nested functions and non-nested functions is that the address of a nested function *N* does not entirely capture *N*'s “value”: the environment is also required. In contrast, the address of a non-nested or global function *G* entirely captures its value, because *G* has no parent and thus needs no environment. The C notion of “pointer-to-function” is therefore sufficient to capture the value of *G* but not that of *N*.

Therefore High C/C++ disallows taking the address of a nested function, and, where C assumes “&” before any expression of type function, High C/C++ does *not* assume the “&” if the expression is of a nested-function type.

## B.7.2 Full-Function Values

We refer to the combination of a function address and its environment as a *full-function value*, as opposed to just a “function address”.

All the capabilities associated with global functions, such as passing them as parameters and storing their values into variables, are available for nested functions, although new syntax is required.

**! declaration** A variable capable of holding a full-function value, and therefore the value of a nested function, is declared as a function declaration would be, except that “!” follows the parenthesized formal parameter list. For example:

```
int ffv()!;
```

In contrast, a ANSI Standard C variable capable of holding only a function address is declared using the pointer syntax:

```
int (*fa)();
```

`ffv` can be called with the expression “`ffv()`”, but *not* with “`(*ffv)()`”, because `ffv` is *not* (just) a pointer.

Any nested function can be assigned to `ffv`. A global function `G` can be assigned to `ffv` by *dereferencing it*, because `G` is transformed to `&G` by the compiler and must be dereferenced to obtain the full-function value of `G`, not just its address: “`ffv = *G;`”. The environment stored in `ffv` in such an assignment is meaningless, since `G` needs no environment. Upon calling the value in `ffv`, the environment is passed to `G`, but `G` (indeed every global function) safely ignores it.

*Passing nested functions as parameters* Nested functions can be passed as parameters: the full value is passed. The full-function value of a global function can be passed by dereferencing the global function; the passed environment is meaningless.

An argument can be declared to be a full-function value by using the new syntax:

```
int f(ffv) int ffv()!; { ... ffv(); ...}
```

*Function constants* Only the names of function constants can be dereferenced to produce full-function values. The dereference of a pointer to a function is immediately converted back to an address by ANSI Standard C rules for function expression conversion.



Thus:

```
extern int sub();
main () {
    int (*fa)();
    int Nested() {...}
    main(*main); // Passes the full value of main
    (*sub);      // Passes the full value of sub
    main(*fa);   // Passes fa, since *fa => &*fa = fa
    main(Nested); // Passes the full value of Nested
}
```

This extension is compatible with ANSI Standard C, because the dereference of an expression of type pointer-to-function is permitted only in the context of an expression denoting a function to be called — for example, “`(*fa)()`”; but “`(*fa)(*fa)`” is not allowed in ANSI Standard C.

*Calling a sort  
function*

As an example of the use of full-function values, we present a call to a sort function that takes as parameters two functions:

```
extern void Quick_sort(
    int Lo, int Hi, int Compare(int a,int b)!,
    void Swap(int a,int b)!
);
static Sort_private_table() {
    Entry Entries[100];
    int Compare(int a,int b) {
        return Entries[a] < Entries[b];
    }
    void Swap(int a,int b) {
        Entry Temp = Entries[a];
        Entries[a] = Entries[b]; Entries[b] = Temp;
    }
    ...
    Quick_sort(1,100,Compare,Swap);
}
```

Here `Compare` and `Swap` must be local to `Sort_private_table` because the table `Entries` is local to that function. In ANSI Standard C, `Entries`, `Compare`, and `Swap` would have to be moved outside of `Sort_private_table`. This works fine in this simple case, but if `Sort_private_table` were recursive, you would have to explicitly manage a stack of `Entries` arrays to get the desired effect.

---

**Note:** A stripped-down version of a practical Pascal program, translated into High C, is included in distributions of High C/C++ compilers. (The name of the file is `analyze.c`.) This program implements a graph traversal algorithm. If you think ANSI Standard C function capabilities are adequate, read this program and try to translate it to ANSI Standard C. In the years since we issued this challenge, no one has taken it and succeeded.

---

*Casting a full-function value* You can cast any full-function value of one type to any full-function value of another type in High C/C++, in concert with the ability to cast function addresses in ANSI Standard C. The `sizeof` a full-function type can be taken and is always greater than the `sizeof` a function address, because the former includes the environment.

*Declaring full-function types* The additional syntax required to permit the declaration of full-function types follows, and is simply a repeat of the rules for ANSI Standard function syntax except that “!” is allowed:

```
Declarator'
-> Extended_function_specification_declarator:
    Declarator' Parameters '!';
Abstract_declarator'
-> Abstract_declarator'? Abstract_parameters '!';
Declarator2'
-> Declarator2' Parameters '!';
```

---

## B.7.3 Non-Local Labels

*Outer labels are visible to nested functions* In High C/C++ a function label is visible to the function in which it is declared and to all of the descendants of that function. A label declared in a descendant overrides the declaration of a label in an ancestor.

Jumping to an ancestor’s label is a valuable technique for easily unwinding a computation deep in the throes of recursion. It is a disciplined form of C’s `setjmp/longjmp`, and comes from Pascal.

For example:

```

void Wind(void Callme(!)) {
    void P(int Cnt) {
        if (!Cnt) Callme();
        P(Cnt-1);
    }
    P(100);
}
void main () {
    void Called() {
        printf("%s%s", "Called from deep within"
               "recursion -- exiting.\n");
        goto EXIT;
    }
    Wind(Called);
    printf("I should not get here.\n"); return;
EXIT:
    printf("%s%s", "Deep recursion unwound by"
           " a single goto.\n");
}

```

Here `main()` invokes a procedure `Wind()` that causes local function `P()` to make 100 invocations of itself before calling function `Called()`. `Called()` disposes of all the invocations of `P()` in one fell swoop by jumping to the `EXIT` label.

The High C/C++ compiler emits the proper code to clean up the stack.

- In locally optimizing High C/C++ compilers, the operation is extremely cheap — just a few instructions — but variables in the function containing the `EXIT` label cannot be allocated to registers.
- In globally optimizing High C/C++ compilers, each unwound stack frame must be visited individually to do the unwinding so that register variables can be used in the `main()` function.

Although this example might seem frivolous, there are serious uses of non-local `gotos` in the High C/C++ compiler itself.

---

## B.8 Intrinsics

High C/C++ contains a set of so-called “intrinsic functions” that supply the ability to take the absolute value, minimum, and maximum of values of any arithmetic type. Intrinsic functions do not need to be declared to be used.

---

## B.9 Packed and Unpacked Structures

Whether High C/C++ aligns structure members by default is platform dependent. Pragma `Align_members` causes alignment for any structure definition occurring within any program region in which this pragma is active (see the **High C/C++ Programmer’s Guide** for information about compiler pragmas and toggles).

Alternatively, you can use the High C/C++ reserved words `_Packed` and `_Unpacked` to specify alignment on a per-structure basis. For example:

```
_Packed struct {char c; int i;} x;           // Packed, as by default
_Unpacked struct {char c; int i;} y;         // Unpacked, not default
struct {char c; int i;} z; // Unpacked, as by default
#pragma Align_members(1);
struct {char c; int i;} w; // Packed, as by default
```

---

**WARNING:** `_Packed` and `_Unpacked` are not implemented for all target platforms. See your `README` file for information.

---

---

## B.10 Near and Far Objects and Pointers

The notions of “near” and “far” are common extensions on architectures whose instruction sets can use addresses of at least two different sizes.

Many objects are addressed relative to a default segment number. Far references specify a non-default segment number in an address reference, and near references specify use of a default segment rather than an otherwise-implied non-default segment number.

Here we do not discuss the specific semantics of near and far, because they are architecture-dependent. Instead we describe the language rules governing them. Consult the **High C/C++ Programmer's Guide** for a full description of the semantics.

High C/C++ introduces the notions of near and far through two new reserved words: `_Near` and `_Far`. The placement of `_Near` and `_Far` follows the same rules as those for `const` and `volatile`.

---

### B.10.1 Distance Attributes

*Any type can be near, far, or huge* Like `const` and `volatile`, any type can possess a near, far, or huge attribute. We call these attributes *distance attributes* and say that a type is *distance attributed*. In what follows, where we refer to the “near” attribute, remember that it is denoted with the reserved word `_Near` (similarly for far and huge); if you prefer using the word `near` itself, use `#define near _Near` in your program, perhaps in the profile `hc.pro`.

*Avoid the ambiguous term “far pointer”* An object with distance-attribute `X` and type `T` can be referred to as having type `X T`; for example, `_Far int`. A pointer to an object with distance-attribute `X` and type `T` can be referred to as having type pointer-to-`X-T`; for example, “pointer-to-far-`int`”. As with `const` and `volatile`, we have seen the imprecise term “far pointer” used to denote, most often, a pointer-to-far-*sometype*. Less frequently it denotes a far-pointer-to-*sometype*. We avoid “far pointer” altogether and use the more explicit “pointer-to-far-...” or “far-pointer-to-...”. With careful wording, English expressions of C attributed types can be unambiguous. For example:

```
int i;                // An int
int *pi;              // A pointer-to-int
_Far int fi;          // A far-int
_Far int *pfi;        // A pointer-to-far-int
int *_Far fpi = &i;   // A far-pointer-to-int
_Far int *_Far fpfi = &fi; // A far-pointer-to-
                        // far-int
_Far int frff();      // A function returning a far-int
_Near int frni();     // A function returning a near-int
int *_Far frfpi();    // A function returning
                        // a far-pointer-to-int
```

```
    _Far int *frpfi(); // A function returning a
                      // a pointer-to-far-int
```

*Attributes of function return types* Functions returning a distance-attributed type seem strange at first glance. What does it mean to return a type with the “far” attribute? We have said that “far” can specify a different addressing mode for an object, but a returned value is not an addressable object.

Rather than ignoring attributes of return types, High C/C++ interprets a function-returning-*X*-type... as a reflection on the nature of the addressing mechanism necessary to call the function, rather than as an implication about its return type. For example, a function-returning-far-*int* is really a “far function”-returning-*int*.

In both the AMD Am29000 and Intel architectures, a function *F* can call function *G* that has the “near” attribute only if *F* is within a certain addressing range encompassing both *F* and *G*; if *G* had the “far” attribute it could be as far away as the architecturally supported maximum instead. See the **High C/C++ Programmer’s Guide** for details.

*Return attributes affect the function, not the return type* Functions *frff*, *frni*, and *frfpi* in the preceding example have attributed return types; the attribute is taken as affecting the function itself. But function *frpfi* does *not* have an attributed return type. Its return type is a pointer type and the base type of the pointer type is that which is attributed. Therefore, just adding the attribute at the leftmost position of a function declaration generally does *not* cause attribution of the function. Placing the attribute nearest to the function name achieves this effect. For example:

```
char * (* _Far F())();
_Far char * (* G())();
```

*F* is a function returning a far-pointer-to-function-returning-pointer-to-*char*, and hence is interpreted as a far function returning a pointer-to-function-returning-pointer-to-*char*. *G* is a function returning a pointer-to-function-returning-pointer-to-far-*char*. *G* itself is not attributed.

---

## B.10.2 Type and Assignment Compatibility

Two types *T* and *T'* that are compatible in the normal sense are also compatible if distance attributed. The same holds for assignment compatibility. Pointer types *T* and *T'* that are normally assignment compatible are also

assignment compatible if either of their base types is distance attributed. However, when values of the two types “meet” in an expression (such as assignment or comparison), conversion code might be required. Conversion is not required for the Am29000 architecture. For Intel architectures, assigning a pointer-to-near to a pointer-to-far requires a conversion because the two types are not the same size. See the **High C/C++ Programmer’s Guide** for the semantics of the conversion.

For two function types, distance attributes figure just the same as the ANSI-specified attributes **const** and **volatile**. Thus a function-returning-far-**int** is *not* compatible with a function-returning-**int**. As a (complicated) example, the cast below is necessary to assign **F** to function variable **FV**:

```
char * (* _Far (*FV)())();
char * (*      F  )();
FV = (char * (*_Far(*)())()) F;
```

Here the cast converts the address of **F** to the same type as that of **FV**. For Intel architectures, code is added to convert the representation of the address of **F** to a value having the same size as that of **FV**’s type. On other architectures such as the Am29000, the distance attribute does not affect the address size and so no conversion code is generated.

Explicit casts of pointer types can also cause conversion code to be generated where the base types have different distance attributes.

---

## B.11 Differences Between High C Syntax and Others

To implement distance attributes, we use the well-studied and thoroughly tested ANSI syntax for **const** and **volatile**. Microsoft C introduces new syntax instead. In Microsoft C, far and near appear in the declarator portion of the syntax and modify the declarator to the right, in their own words. Thus Microsoft C will not accept:

```
far int x;
```

but instead requires:

```
int far x;
```

This is parsed as:

Specifier	Declarator
<code>int far</code>	x

High C/C++ accepts either. They are both parsed as:

Specifier	Declarator
<code>_Far int</code>	x
<code>int _Far</code>	x

Microsoft C and High C/C++ syntax are incompatible when the declarator involves both left- and right-associative operators. For example, in High C/C++ declare a pointer to a far function with:

```
_Far void (*f)();
```

Here the function pointer returns a “far void” type, which causes the function itself to be regarded as far, and its return simply as “void”. Microsoft C requires:

```
void (far *f)();
```

where `far` modifies the declarator `*f`. The High C/C++ solution is better because it avoids introducing the notion of “modified declarator,” a notion that is not in the language. Instead, we use the existing ANSI-specified rules for the placement of `const` and `volatile`.

---

## B.12 Predefined Macro `__HIGHC__`

This macro is predefined to be `1` (one) for all implementations of High C/C++, unless you are compiling in ANSI mode. You can use `__HIGHC__` to exclude code that depends upon non-ANSI-supported features of High C/C++.



## B.13 Calling-Convention Attribute Specified with Function Declaration

In addition to the normal High C/C++ way of using the calling-convention pragma, we have introduced a new calling-convention specification syntax that makes it easier to compile programs acceptable to Microsoft C — specifically, Windows applications, which contain lots of mixed-model programming.

The syntactic construct:

```
_CC( E )
```

where *E* is an arbitrary expression, is permitted where **const**, **volatile**, **\_Near**, and **\_Far** are permitted. *E* must be a calling-convention expression, of the same sort permitted in the `Calling_convention` pragma. For example:

```
_CC( _CALLEE_POPS_STACK | _SAVE_REGS )
```

specifies the Microsoft Pascal calling convention: the callee pops the stack, non-volatile registers are saved if the routine uses them, and parameters are pushed from left to right (right to left requires the `_REVERSE_PARAMS` attribute).

The return type of a function, if it has `_CC` attributes, reflects those attributes to the function being defined. For example:

```
#define MSPASCAL _CC( _CALLEE_POPS_STACK | _SAVE_REGS )
int _Far MSPASCAL f() { ... }
```

`f` is a function returning a “far” `int` with calling convention `_CALLEE_POPS_STACK | _SAVE_REGS`. The attributes of the return type reflect back onto the function being defined, so `f` is really a “far function having calling convention `_CALLEE_POPS_STACK | _SAVE_REGS` and returning an `int`”.

Multiple `_CC( . . . )` specifications can appear in a single declaration; the effect of `_CC(e1) _CC(e2)` is the same as that of `_CC(e1 | e2)`.

There is a restriction: the `_CC` syntax applies only in the presence of a function declarator.

The following does *not* work:

```
typedef MSPASCAL x;
x f();
```

The MSPASCAL is lost.

The set of `_CC` attributes is cumulative in the specifiers. That is, if more than one `_CC` is specified, the final convention is the bit-wise OR of all the constituents. A calling-convention specified by `_CC` *completely overrides* that given by a calling-convention pragma.

**Caveat** The calling convention present on a declaration of a function overrides the convention present on a subsequent definition, without any warning from the compiler. Thus, in:

```
void MSPASCAL f();
void f() { ... }
```

`f` will have the MSPASCAL calling-convention. In:

```
void f();
void MSPASCAL f() { ... }
```

`f` will *not* have the MSPASCAL calling-convention.

---

## B.13.1 Kanji Support

*Double-byte Kanji characters* The High C/C++ compiler supports double-byte Kanji characters in C strings. Kanji support requires 6K more memory to read alternate scan tables. [The High C Library](#), especially the string functions, does *not* support Kanji.

The Kanji support consists of a different interpretation of a C string. In reading a C string from left to right, any occurrence of a character `C1` in the range `0x80..0x9f` or `0xe0..0xfc` is considered to be the first character of a two-byte Kanji character. The character `C2` that follows `C1` is taken without interpretation. Thus `C2` could be `\`, without confusing it with the standard C escape sequence.

*Japanese versus ANSI Standard* This method of Kanji support is a *de facto* Japanese standard. The ANSI Standard dictates that character strings of the form `L"xxxx"` denote multi-byte character strings.

The **High C/C++ Programmer's Guide** discusses compiler options for Kanji support.

---

## B.14 short enum and long enum

In High C/C++, you can use the adjectives **short** and **long** in declarations of **enum** types. The former indicates that the enumeration type should use as little space as possible. The latter indicates that the enumeration type should have size equal to **sizeof(int)**. The former case is the default unless overridden by the toggle **Long\_enums** (see the **High C/C++ Programmer's Guide** for information about compiler toggles). For example:

```
short enum se {red,white,blue}; // sizeof == 1
long  enum le {alpha,beta,gamma}; // sizeof ==
                                   // sizeof(int)
```

---

## B.15 Arbitrary Base for Numbers

In High C/C++, you can specify any base (radix) from **1** (one) to **16** for an integer constant, whereas ANSI Standard C supports only octal, decimal, and hexadecimal integer constants.

*Switching to a new  
base*

Preprocessor numbers allow High C/C++ to recognize a second or later **x** in a number beginning with **0x** as a switch to a new base for interpreting the values after the **x**. The currently evaluated number becomes the value for the new base. In this way you can specify binary numbers or any other base up to **16** in High C/C++:

```
0xfffe           // As usual
0x2x1001         // Decimal 9 (binary number)
0x2x1001x20      // Base 9 number with value 18
                 // (20 base 9)
0x10xfffe        // Same as 0xfffe since 0x10 is 16
```

## B.15.1 ANSI Preprocessor Numbers

When ANSI Standard C *preprocessor numbers* were implemented in High C the lexical syntax for a number changed to:

```
'.'? Digit (Letter|Digit|'.'|('e'|'E')('+'|'-'))*
```

This means, for example, that `3` and `3.3.f.e+e-` are both valid preprocessor numbers.

*Using token pasting  
to construct  
numbers*

Preprocessor numbers allow you to construct numbers out of their constituents with token pasting; for example, pasting `3e` to `10` results in the floating-point number `3e10` (see §4.4: *Concatenation*). After all preprocessing is done the compiler diagnoses any preprocessor numbers that are not valid numbers; so the `3.3.f.e+e-` example given above will be diagnosed if it is not removed from the token stream during preprocessing, either by simple deletion or by converting it to a string.

*Range syntax  
.. not allowed*

Preprocessor numbers invalidate the High C/C++ extension `..` that has been present in High C/C++ since its inception. For example, `case 3..5:` is now a single case with the value `3..5`, which is a valid preprocessor number but an invalid number. Therefore, to protect the original `..` extension, High C/C++ does not allow `..` in a preprocessor number except when compiling in ANSI mode.

This extension is disabled when you compile in ANSI mode.

## B.15.2 Preprocessor Directives `#assert` and `#unassert`

Preprocessor directive `#assert` allows you to associate a value with a name that you can subsequently reference in a `#if` statement, without affecting the namespace of macros. You use the `#assert` statement to define a *predicate* (a preprocessor construct) by associating with it a token sequence:

```
#assert predicate(token_sequence)
```

For example:

```
#assert machine(386i)
#assert os(unix)
```

You test for the assertion as follows:

```
#if #predicate(token_sequence)
```

For example:

```
...  
#if #machine(386i) && #os(unix)  
...  
#endif
```

To remove all assertions against a predicate:

```
#unassert predicate
```

To remove a single assertion:

```
#assert predicate(token_sequence)
```

**#assert** and **#unassert** are supported in all globally optimizing High C compilers; however, they are considered useful only on UNIX System V Release 4.



# C

# Iterators in High C/C++

---

This [appendix](#) describes a powerful language construct called an *iterator*. Iterators allow you to express some computations more naturally, by virtue of recursion. They can make your code easier to maintain, because there are fewer places where code must be modified if there is a design change.

---

## C.1 What Is an Iterator?

An *iterator* is a new kind of function that supplies values for the **for** variable(s) of an *iterator-driven for loop*. The **for** loop body is executed each time the iterator produces value(s) for the **for** variable(s) via a new pre-defined function `yield()`. Recursion is possible with iterators because the function that drives the **for** loop does not have to return.

- Benefits of iterators* The major benefit of iterators and iterator-driven **for** loops is that the algorithm to determine the values of the **for** variables for each loop iteration is defined separately in an “iterator” rather than being exposed in the ANSI Standard C **for**-loop construct itself. The iterator can be invoked in many such **for** loops.
- Change data access without changing loop code* If you use iterator-driven **for** loops to access the elements of a data structure, iterators allow you to change the implementation of the data type without changing the implementation of the **for** loops.
- With an iterator, each **for** loop is driven by an independently specified computation that can maintain its environment across invocations of the loop.
  - Without an iterator, the syntax accessing the structure is *inside* each **for** loop; and therefore each **for** loop must be modified, because the loops contain code specific to the data structure.
- Origin of iterators* Iterators originated in CLU, a programming language from MIT that was designed to promote the use of abstractions in program construction. Iterators and iterator-driven **for** loops are not part of ANSI Standard C. They have

been made available in High C/C++ because they help programmers produce readable, easily modified code.

## C.2 How Do Iterators Work?

In ANSI Standard C you might write:

```
int i;
for (i = 1; i <= 10; i++)
    printf("%d squared is %d\n", i, i*i);
```

Here is how you could write this loop using an iterator:

```
void Upto(int Lo, int Hi) -> (int) {
    int i = Lo;
    while (i <= Hi) yield(i++);
}
for i <- Upto(1,10) do
    printf("%d squared is %d\n", i, i*i);
```

The sequencing from one number to another (1 (one) through 10 in this example) is programmed once in the `Upto` iterator. The syntax `for i <- Upto(1,10)` “starts” the iteration.

*Predefined function*  
*yield* Each time the iterator `Upto` calls the predefined function `yield()` with a value, that value is substituted for `i` in the body of the `for` loop, and the body is executed. When execution of the body is finished, control is returned to a point immediately after the `yield()` and `Upto` continues its `while` statement.

Each call to `yield()` causes the `for`-loop body to be executed once. When `Upto` is finished `yielding`, it returns just like a regular C function, and the invocation of `Upto` (and therefore the `for` loop itself) is complete. Control then passes to the statement after the `for` loop.

*Special iterator*  
*syntax ->* The syntax `->(int)` in the header of the iterator definition is the only thing that distinguishes it from a normal function. After the `->` appears a prototype-form parameter list specifying the type(s) of the results `yielded` by the iterator (more on multiple `yields` later). Here `Upto` `yields` an `int` — one for each execution of the `for`-loop body. *Within* the definition of an iterator, the predefined function `yield()` is defined. *Outside* an iterator you can use the name `yield()` for any other purpose.



## C.3 Some Uses for Iterators

This section presents some programming situations where you might want to consider using iterators instead of regular **for** loops, and introduces a few of the programming problems that can be solved with iterators.

*Iterating over the elements of a set*

Although iterators are quite general, they are probably most useful for iterating over each element in some set. The set is often the entire contents of some data structure, but can be restricted to those elements in a data structure that meet certain conditions. The set might not be stored in a data structure at all; for example, an iterator could be used to provide the prime numbers up to 100 by computing and **yielding** them.

*List processing*

Here is a loop that processes each element in a list:

```
for e <- each_element(list) do {
    ... // Process element e
}
```

In the loop, **e** is the single **for** variable. Each time through the loop, **e** is given a value yielded by the iterator **each\_element**. (In this case, we would expect **e** to have a different value each time through the loop, but this is not always so.) **e** is really a parameter to the loop body. It is not visible outside the loop body and is instantiated anew each time through the loop. Assuming the following declaration of an **element**:

```
typedef struct element {
    ... // Some data
    struct element *next;
} *element;
```

the iterator **each\_element** might look like this:

```
void each_element(element e) -> (element) {
    while (e != 0) {
        yield(e);
        e = e->next;
    }
}
```

Each time the predefined function **yield()** is called, control passes to the **for** loop that invoked **each\_element**. The **for** variable **element** takes on the current value of **e** for that pass through the loop body. At the bottom of

the **for** loop, control returns to `each_element` at the statement following the call to `yield()`.

*Changing the  
implementation of a  
data type*

Suppose in some program you make heavy use of a sorted linked list. If you want to speed up your code, you might replace the list with a binary tree. That would mean not only replacing the code that *implements* the list (the insert, delete, and sort functions, for example) but also finding all the places in the program where the list gets *traversed*. You would need to replace all the **for** loops that are some variation on:

```
for (listptr = head; listptr != NULL;
     listptr = listptr->next) {
    ... // Uses list element
}
```

You might traverse the list several places in your program, and you would have to change them all. On the other hand, if you use an iterator, you would still have to replace the code that *implements* the list, but you would not have to touch the code that *traverses* the list. Your loops would look something like this (both before and after you changed over to a binary tree):

```
for listptr <- traverse_list(head) do {
    ... // Uses list element
}
```

*Traversing a tree*

A **for** loop is a natural way to conceptualize getting each item in a data structure. You might write pseudo-code for some loop that processes every node in a tree, such as:

```
for each node in tree {
    ... // Processes node
}
```

However, when you actually write such a loop in C, you would probably not use a **for** loop. Trees are naturally traversed recursively, and C's **for**-loop construct cannot naturally express a recursive computation. However, a High C/C++ iterator-driven **for** loop can express recursive computations as easily as any other kind; the resulting code looks almost identical to the pseudo-code:

```
for n <- each_node(tree) do {
    ... // Processes n
}
```

The iterator `each_node` can be recursive. Thus, a recursive algorithm can easily be used to generate values for variables.

---

## C.4 Recursion and Code Clarity

*Iterating through the nodes of a binary tree* In the list-traversal examples in §C.3: *Some Uses for Iterators* on page 281, you might complain that all we have done is make nice, straightforward **for** loops complicated. (We have done more than that: now the list implementation can be changed without changing the **for** loops.) But consider a different problem: iterate through the nodes of a binary tree, where the tree is implemented using a data structure that for each node **N** has a pointer to the left and right subtrees of **N**.

*A recursive tree walk* The obvious way to obtain the nodes of such a tree is by a simple recursive tree walk. However, this is not possible using ANSI Standard C **for** loops. Imagine having to translate the following pseudo-code into C:

```
Process the tree:
    do some stuff to the tree
    for each node in the tree
        print the node
    do some other stuff to the tree
```

Because you cannot use a **for** loop to get the nodes of the tree, you would probably package the node-printing process into a routine and pass that routine to be executed by a tree-walking routine:

```
typedef struct node {
    ... // Some data
    struct node *Left, *Right;
    // Left and right subtrees
} *Node;

void Print_node(Node N) {
    printf("Node is ");
    ... // Code to print a Node
}

void Each_node(Node N, void Doit_toit(Node N)) {
    if (N == 0) return;
    Each_node(N->Left, Doit_toit);
```

```

    Doit_toit(N);
    Each_node(N->Right, Doit_toit);
}
Process_the_tree(Node Root) {
... // Does some stuff to the tree
    Each_node(Root, Print_node);
... // Does some other stuff to the tree
}

```

Rearranging the computation this way allows the recursive tree walk to occur, but there is a cost: the simple **for** loop in the pseudo-code clearly expresses that a computation is being done on each element of the tree data structure; this is no longer as apparent in the body of `Process_the_tree`.

To find out what is being done you now have to look *outside* `Process_the_tree`, in `Print_node`. In this simple example, the choice of good names assists a great deal in promoting understanding of the code, but when dealing with real problems, the code is more complex and the names are usually not perfectly explanatory.

*Using iterators for  
the tree walk*

Here is a solution to the same problem, using iterators:

```

void Each_node(Node N) -> (Node) {
    if (N == 0) return;
    // Walks the subtrees
    for L <- Each_node(N->Left) do yield(L);
    yield(N);
    for R <- Each_node(N->Right) do yield(R);
}
Process_the_tree(Node Root) {
... // Does some stuff to the tree
    for Node <- Each_node(Root) do {
        printf("Node is ");
        ... // Code to print a Node
    }
... // Does some other stuff to the tree
}

```

Notice that the code for `Process_the_tree` mirrors the pseudo-code extremely closely, yet unlike the ANSI Standard C version, the algorithm that determines the successive values of `Node` in the **for**-loop body is *recursive*. This is a major increase in expressive power. We are not claiming an increase in computational power. C is already Turing-machine equivalent. We are

claiming that algorithms can be expressed in a more natural fashion, making the code easier to write, understand, and modify.

The body of `Each_node` is not very elegant (or efficient) as written. The recursive calls do nothing but re-`yield` a result already `yielded` at a deeper level of recursion.

*Functions nested  
within iterators*

Using nested functions (another High C/C++ extension) and the fact that calls to `yield()` can occur within functions nested within iterators, we can produce an elegant version:

```
void Each_node(Node N) -> (Node) {
    void P(Node N) {
        if (N == 0) return;
        // Walk the subtrees.
        P(N->Left);
        yield(N);
        P(N->Right);
    }
    P(N);
}
```

*Let the compiler do  
the work*

Work done by the *programmer* in the ANSI Standard C version — packaging the body of the pseudo-code `for` loop as a function — is done instead by the High C/C++ *compiler* in the iterator version. This is appropriate; after all, compilers were invented to promote more easily understandable and modifiable languages. Why make a programmer do what the compiler can do?

---

## C.5 Replacing Macros with Iterators

Another place an iterator can be gainfully used is when the iteration algorithm, although not recursive, is complex. Often in this circumstance you would design a macro to make up for the shortcomings of the C `for` loop.

*Sequencing  
through  
overlapping objects*

Consider the following example, taken from the High C/C++ compiler. The problem is to sequence through the objects that overlap a given object `obj` in memory. Prior to the addition of iterators, a macro was required to simulate the iteration.

```

#define for_each_overlapping_object(o,obj,p){\
    struct obj_entry *_op = &objtab[obj];\
    obj_class_type _class = _op->class;\
    long _len = _op->len;\
    long _disp = _op->disp;\
    ushort _word = _op->un.word;\
    bool is_deref = (ea_DEREF & _op->flags) != 0;\
    bool is_adr = ((ea_ADR|ea_GLOBAL)\
        & _op->flags)!= 0;\
    register struct obj_entry *p; int o;\
    for(o=1,p=objtab+1;o<=last_object;o++,p++){
if (is_deref && \
    ((p->flags&ea_DEREF) || \
    (p->flags&(ea_ADR|ea_GLOBAL))!=0 &&\
    p->xlen >= _op->xlen &&\
    _op->disp < p->xlen) || \
    is_adr && (p->flags&ea_DEREF)!=0 &&\
    p->xlen<= _op->xlen || \
    p->class == _class && \
    (p->un.word == _word || \
    (p->flags&_op->flags&ea_TEMP)) &&\
    (p->disp<=_disp && p->disp+p->len > || \
    _disp+_len > p->disp && _disp < p->disp)){

```

`obj_class_type` is defined elsewhere; due to data abstraction, in the preceding example you do not need to know exactly what an `obj_class_type` is. Given an `object_index`, the macro in the example iterates over each object overlapping with that `object_index`. The macro is invoked as follows:

```

object_index obj;
...
for_each_overlapping_object(o,obj,p)
    ... do things with o and p ...
    }}} // Required to match {'s in macro

```

The `#define` defines the variables `o` and `p` whose names are arguments to the macro.

*Reducing  
maintenance and  
code size*

With this approach, an enormous amount of code is reproduced each time the macro is invoked; and every time a change is made to the macro, every module where it appears must be recompiled. An iterator requires much less

maintenance: when a change is made, only the iterator itself needs be recompiled, not its “client” **for** loops.

*Improved type  
checking*

Another advantage is that the iterator can declare the types of its parameter and **yielded** results, providing improved type checking at the iterator invocation. The following example shows the same routine implemented as an iterator:

```
void Each_overlapping_object(object_index obj)
-> (int o, struct obj_entry *p) {
    struct obj_entry *_op = &objtab[obj];
    obj_class_type _class = _op->class;
    long _len = _op->len;
    long _disp = _op->disp;
    ushort _word = _op->un.word;
    bool is_deref = (ea_DEREF & _op->flags) != 0;
    bool is_adr = ((ea_ADR|ea_GLOBAL) & _op->flags)
        != 0;
    register struct obj_entry *p; int o;
    for(o=1,p=objtab+1;o<=last_object;o++,p++){
        if (is_deref &&
            ((p->flags&ea_DEREF) ||
             (p->flags&(ea_ADR|ea_GLOBAL))!=0 &&
             p->xlen >= _op->xlen &&
             _op->disp < p->xlen) ||
            is_adr && (p->flags&ea_DEREF)!=0 &&
            p->xlen<= _op->xlen ||
            p->class == _class &&
            (p->un.word == _word ||
             (p->flags&_op->flags&ea_TEMP)) &&
            (p->disp<=_disp && p->disp+p->len > _disp ||
             _disp+_len > p->disp && _disp < p->disp)) {
            yield(o,p);
        }
    }
}
```

The iterator in this example is invoked as follows:

```
object_index obj;
...
for o,p <- Each_overlapping_object(obj) do
    ... do things with o and p ...
```

The iterator `yields` *two* results for each execution of the `for`-loop body: the `int o` and the `struct obj_entry *p`.

---

## C.6 Iterator Syntax and Constraints

To invoke an iterator `I` for `n` `for` variables, write:

```
for Name1, Name2, ... Namen <- I(E1, E2, ... Em) do
...    // Body of for loop, using the Namei
```

Iterators yield one or more values. The declarative syntax for both the iterator input arguments and `yield()` types follows that of the input arguments to normal functions. An iterator `I` is declared as follows:

```
void I(Formal_parameter_list1) ->
(Formal_parameter_list2) {
...    // Body of iterator, with calls to yield
}
```

---

### C.6.1 Arguments to Iterators

*Prototype syntax is required* The iterator's `Formal_parameter_list1` has exactly the same constraints and semantics as the formal parameter list of other functions, except that we additionally require that this list use the ANSI Standard C prototype syntax. Old-style C function parameter declarations are not permitted. When the iterator is invoked in a `for` loop, the expressions passed to it must satisfy the same constraints as those of expressions passed to a function, given the same parameter list, and the parameters are passed in the same ways.

The `Formal_parameter_list2` must also use the ANSI Standard C prototype syntax. Here, however, the names of the parameters can be omitted, just as they can be omitted when a function is declared but not defined.

*The yield list* The `Formal_parameter_list2` is called the *yield list*. The `for` variables of a `for` loop that invokes the iterator take on values of the types specified in the yield list, respectively. The iterator supplies the values through a call to the function `yield()` which is defined in the body of each iterator:

```
yield(E1, E2, ... En);
```



where the following constraint must be satisfied:

If `void yield( Formal_parameter_list2 );` is a valid function declaration, then `yield(E1, E2, ... En);` must be a valid call to that function.

*Using named parameters*

Alternatively, the syntax of High C/C++'s named parameter association can be employed to yield the values, just as in a function call. The `Formal_parameter_list2` must contain the names of the parameters (see §B.6: *Named Parameter Association* on page 261). However, even if that notation is not used, it is useful to name the “yield parameters” for the sake of documentation.

*Specifying a variable number of arguments*

Like other functions, iterators can use the `...` notation to specify that they can take a variable number of arguments. The `yield()` formal-parameter list can also use `...`, requiring the use of the `va_arg` macros within the `for`-loop body to access the remainder of the expressions `yielded`.

---

### C.6.2 Predefined Function `yield()`

Because the types `yielded` by an iterator are described by a `Formal_parameter_list`, there is no restriction on the types except that they must be types that can be passed to a normal function. Thus, an iterator can `yield` integers, structures, functions, and even iterators. An iterator that computes the strongly connected components of a graph and an example of its use are provided in High C/C++ distributions in file `analyze.c`; this iterator's `yield` is itself an iterator that yields one set of strongly connected components. No one has yet accepted our challenge to rewrite this example in ANSI Standard C.

---

## C.7 Semantics of Iterators

Each time an iterator executes a `yield()` statement, the body of the `for` loop is executed with its `ith` `for` variable assuming the value of the `ith` expression `yielded`, for all `1 <= i <= m`. When the iterator returns (just as a function can return), the `for` loop is terminated. A `break` or `goto` from the `for`-loop body also terminates the iteration.

The semantics of iterators and **for** loops that invoke them can be specified precisely in terms of an implementation using nested functions, as follows.

*for loops are bundled as functions and passed to the iterator*

The compiler bundles up the body of each iterator-driven **for** loop and turns it into a function. Each so-bundled function is passed as an extra parameter to the iterator; the iterator is called once per **for** loop and receives as a parameter the function that used to be the body of that **for** loop. Each call to **yield()** within the iterator is translated to a call to the function that is its extra parameter.

More formally, the meaning of:

```
void I(Formal_parameter_list1) ->
    (Formal_parameter_list2);
...    // Calls to yield in here
}
```

and:

```
for Name1, Name2, ... Namen <- I(E1, E2, ... Em) do
...    // for-loop body here
```

is precisely the same as the meaning of:

```
void I(void yield(Formal_parameter_list2)!,
        Formal_parameter_list1) {
...    // Calls to yield in here
}
```

and:

```
{// Would-be for loop
void For_loop_body(Formal_parameter_list3) {
...    // for-loop body here
}
I(For_loop_body, E1, E2, ... Em);
}
```

where *Formal\_parameter\_list3* is *Formal\_parameter\_list2* but with the names *Name<sub>1</sub>*, *Name<sub>2</sub>*, ... *Name<sub>n</sub>* replacing the parameter names (if given) in *Formal\_parameter\_list2*.

The calls to **yield()** in the iterator **I** become calls to function **I**'s function parameter named **yield** — the syntax is the same in both cases. Also, the **for**-loop body is made into a function and passed as an “extra” first param-

ter to the function `I`. Calling function `I` is what starts the loop; `I`'s return terminates the loop, barring a `goto` out of the loop body.

It is the function `I` *per se* that does the iterating. That can occur via a contained loop, or by `I` calling itself recursively as it traverses some data structure, for example. Each time it wants to “yield” something to the `for`-loop body, it does so by calling its `yield` parameter and passing the value(s) of the `for` variable(s) to the next iteration of the loop.

Use `!` to pass a nested function as a parameter

The `!` declaration guarantees we can pass a nested (non-global) function to the iterator. Because the `for` loop is guaranteed to lie within a function, the `for`-loop body, when transformed into a function, is guaranteed to be a nested function. For example:

```
void Primes(int Lo,int Hi) -> (int ThePrime) {
    // We named the yield result.
    // Yields the primes in the interval Lo..Hi
    int I,J;
    extern double sqrt(double);
    for (I = Lo; I <= Hi; I++) {
        // Asks if we can divide I evenly:
        for (J = 2; J <= sqrt(I); J++)
            if ((I/J)*J == I) goto Composite;
        yield(I); // I is prime.
    Composite: ;
    }
}

...
for I <- Primes(1,100)
do printf("%d is prime.\n",I);
```

This example has the same semantics as the following:

```
void Primes(void yield(int ThePrime)!,
            int Lo, int Hi) {
    // Yields the primes in the interval Lo..Hi
    int I,J;
    extern double sqrt(double);
```

```

    for (I = Lo; I <= Hi; I++) {
        // Ask if we can divide I evenly:
        for (J = 2; J <= sqrt(I); J++)
            if ((I/J)*J == I) goto Composite;
        yield(I);                                // I is prime.
    Composite: ;
    }
}
...
{
    void For_loop_body(int I) {
        printf("%d is prime.\n",I);
    }
    Primes(For_loop_body,1,100);
}

```

---

## C.8 Advantages and Disadvantages of Iterators

You must balance the advantages of reusable code and lower maintenance against the reduced execution speed that can result from using iterators.

*Reusable code* The major benefit of iterators is that the algorithm to determine the values of the **for** variables for each loop iteration is defined separately in the iterator. Unlike the ANSI Standard C **for** loop, the iterator can do an arbitrary amount of computation and can automatically maintain its environment across passes through the body of the loop, because it suspends rather than returns when it provides the loop body with values.

Once an iterator is written, it can be invoked by as many loops as desired without repeating the code that provides the values used within the loop (as compared with using many copies of standard **for** loops, which involve repeating the code for each loop).

*Information hiding* The structured-programming term *information hiding* refers to placing information only where it is needed, and keeping information from where it is not needed. Information hiding promotes better and more easily maintained programs, and reduces recompilation of modified programs. Iterators promote information hiding.

For example, in a module implementing a tree data type, you might want to hide all the details of traversing trees within functions defined in that module. In ANSI Standard C, however, traversing the tree generally means exposing the tree data structure in any **for** loop sequencing through nodes of the tree. Instead, you can use iterators to localize the sequencing techniques and their required data-structure access to within the tree module itself. The **for** loop only needs to invoke an appropriate iterator and does not need to be concerned with the representation of trees.

*Execution overhead* The major drawback of iterators is that they are slow, compared to standard **for** loops, because they involve the overhead of function calls. Speed is lost by placing the computation of the iteration in a separate function. A loop that executes  $n$  times involves  $n+1$  function calls related to loop overhead. The speed loss occurs not only due to the function linkage but also because the body of the **for** loop becomes a nested function, and any variables in the body's parent accessed from the body cannot reside in high-speed machine registers, due to the current state of optimizer technology. For example:

```
int Sum = 0;
for I <- Upto(1,10) do
    Sum += I;
printf("Sum of 1 to 10 is %d\n", Sum);
```

This is implemented as:

```
int Sum = 0;
void Body(int I) {Sum += I;}
Upto(Body,1,10);
printf("Sum of 1 to 10 is %d\n", Sum);
```

Here, **Sum** cannot reside in a register because it is accessed from nested function **Body**. Thus all accesses to **Sum** will be slowed. Therefore, if extreme speed is required, do not use iterators, especially for simple **for** loops where the iteration algorithm is simple and there is no need for “hiding” it away in one place in an iterator.



# Index

---

logical NOT operator	!	23, 88
	! declaration	291
	! following parameter list for full-function or nested-function values	264
relational operator	!= (not equal to)	83
punctuator	","	16
punctuator	".."	16
punctuator	"..."	16
punctuator	":"	16
punctuator	";"	16
arguments to operators	# and ##	42
uses of operators	# and ##	43
remainder operator	%	82
address operator	&	87
bitwise AND operator	&	85
logical AND operator	&&	85
punctuators	( and )	16
postfix function-call operator	()	202
dereference operator	*	23, 87
multiplication operator	*	82
addition operator	+	81
unary plus operator	+	23, 87
post-increment operator	++	23, 89
pre-increment operator	++	23, 88
postfix unary operators	++ and --	201
comma operator	,	81
post-decrement operator	--	24, 89
postfix unary operators ++ and	--	201
pre-decrement operator	--	24, 88
pointer-to-member operators .* and	->	86
postfix dereference operator	->	203
special iterator syntax	->	280
pointer-to-member operator	->*	86
pointer-to-member operators	.* and ->	86
division operator	/	82
comment style	/**/	17
comment style	//	17, 33
scope resolution operator	::	49, 55, 73, 130, 145, 147
relational operator	< (less than)	83
leftshift operator	<<	84
relational operator	<= (less than or equal to)	83

assignment operator	=	79, 186
binary operator	=	201
operator	=()	164
relational operator	== (equal to)	83
relational operator	> (greater than)	83
relational operator	>= (greater than or equal to)	83
right shift operator	>>	84
ternary operator	?:	78
punctuators	[ and ]	16
data type char	[]	31
data type wchar_t	[]	31
postfix subscript operator	[]	202
line splice character	\	33
backslash character	\ in escape sequences	30
bitwise exclusive OR operator	^	85
punctuators	{ and }	16
bitwise OR operator		85
logical OR operator		85
bitwise NOT operator	~	23, 88

## A

	abstract classes	14, 146
	abstractions in program construction	279
public	access implied by struct or union	153
	access specified for a base class	153
	access_specifier	124, 138, 152
	accessible base classes	86
	active exceptions	227, 231
stack discipline of	active exceptions	231
formal and	actual parameters	116
programming language	Ada	261
	Ada packages	215
	addition and subtraction	81
	addition operator +	81
	additive operators	81
	address of a nested function	263
	address of an overloaded function	87
	address operator &	87
function	address versus full-function value	264
casting function	addresses	266
	addresses as function parameters	118
up-level	addressing	263
High C/C++ extensions: initializing automatic	aggregates	260
High C/C++ extensions: type qualifier	_Alias	19, 69
namespace	aliasing definitions	217



pragma	Align_members - set the maximum boundary for structure-member alignment .....	268
controlling the	alignment of structure members .....	63
	allocating a data item to a register .....	51
	allocating an object of array type .....	91
	allocating dynamic memory with operator new .....	91
	allocating storage for an object .....	91
storage	allocation for direct bases .....	140
Incremental C++	alternate keywords .....	254
	alternate scan tables .....	274
C++ constructs that introduce parsing	ambiguities .....	255
	ambiguities between user-defined operators and pre-defined C++ operators .....	182
resolving	ambiguity .....	101
	AMD and Intel architectures .....	270, 271
	American National Standards Institute .....	1
source file	analyze.c .....	266, 289
bitwise	AND operator & .....	85
logical	AND operator && .....	85
The	Annotated C++ Reference Manual by Margaret Ellis and Bjarne Stroustrup .....	1
	anonymous unions .....	65, 188
differences between	ANSI C and C++ .....	1
	ANSI C offsetof macro .....	149
	ANSI C++ standardization .....	1, 251
compiling in	ANSI mode .....	272, 276
	ANSI preprocessor numbers .....	276
	ANSI prototype syntax .....	288
	ANSI X3J16 standard .....	1, 209
High C/C++ extensions:	arbitrary bases for numbers .....	29, 275
AMD and Intel	architectures .....	270, 271
parameters	argc and argv .....	114
the exception-handler	argument .....	227
hidden	argument this .....	13, 134, 135, 136, 141, 147, 165
	argument type matching .....	194
overloaded functions with the same	argument types .....	218
	argument-match overload resolution .....	192
	argument_decl_list .....	111, 112
	argument_declaration .....	112, 116
assignment operators accepting const	arguments .....	188
constant expressions as template	arguments .....	207
iterator input	arguments .....	288
template	arguments .....	206
variable number of	arguments .....	289
	arguments to constructors .....	92, 166
	arguments to iterators .....	288

	arguments to operator new .....	91, 92, 256
	arguments to operators # and ## .....	42
parameters argc and	argv .....	114
	arithmetic conversions .....	94
	arithmetic data types .....	56
implementing a multi-dimensional	array .....	202
implementing an associative	array .....	203
	array bounds .....	79
	array elements .....	61
	array expressions .....	62
	array of char .....	31
	array of wchar_t .....	31
	array subscripts .....	61
allocating an object of	array type .....	91
	arrays .....	61
declaring	arrays .....	61
deleting	arrays .....	90
initializing	arrays .....	72
multi-dimensional	arrays .....	62
	arrays as function parameters .....	118
preprocessor directives	#assert and #unassert .....	276
compiler-generated	assignment .....	188
pointer	assignment .....	60
type and	assignment compatibility .....	270
type of an	assignment expression .....	80
generated default	assignment operator .....	188
special default	assignment operator .....	188
topmost	assignment operator .....	188
	assignment operator = .....	79, 186
	assignment operators accepting const arguments .....	188
High C/C++ extensions: named parameter	association .....	261, 289
named parameter	association suffix .....	262
implementing an	associative array .....	203
left- and right-	associative operators .....	272
	associativity of operators .....	20
calling-convention	attribute _CALLEE_POPS_STACK .....	273
calling-convention	attribute _REVERSE_PARMS .....	273
calling-convention	attribute _SAVE_REGS .....	273
specifying the calling-convention	attribute with function declaration .....	273
	attributed return types .....	270
distance	attributes .....	269, 271
	attributes of an identifier .....	47
storage categories	auto, extern, register, and static .....	50

## B

	backslash character \ in escape sequences .....	30
	base .....	138
access of a		
access of a public or protected name		
inherited from a	base class .....	157
access specified for a	base class .....	153
access to protected members of a	base class .....	156
declaring a class to be a	base class .....	138
implicit conversions from a derived class to		
a	base class .....	239
inheritance of members from a	base class .....	139
members of the	base class inherited as members .....	153
conversions from a	base class to a derived class .....	239
using dynamic_cast to make conversions from	base class to pointer class .....	242
accessible	base classes .....	86
derived and	base classes .....	137
multiple access through virtual	base classes .....	158
virtual	base classes .....	146
virtual	base classes and constructors .....	172
numeric	base conversion .....	29
virtual	base pointers .....	147
	base_list .....	124
	base_spec .....	124
	base_specifier .....	124
	bases .....	140
direct and indirect	bases .....	140
storage allocation for direct	bases .....	138
virtual	bases .....	146
virtual functions versus virtual	bases and members .....	171
order of initialization of	bases for numbers .....	29, 275
High C/C++ extensions: arbitrary	binary numbers .....	275
	binary operator = .....	201
	binary operators .....	24, 82, 201
iterating through the nodes of a	binary tree .....	283
late	binding .....	13
	bit fields .....	66
	bit-field declaration .....	67, 130
	bit-field lengths .....	79
static	bit-field members .....	132
	bitwise AND operator & .....	85
	bitwise AND, OR, and exclusive OR .....	85
	bitwise copy .....	176
	bitwise exclusive OR operator ^ .....	85
	bitwise NOT operator ~ .....	23, 88
	bitwise OR operator   .....	85

	blank spaces .....	16
declaring a class within a	block .....	127
compound statements or	blocks .....	101
inner	blocks .....	49
	blocks and name scoping .....	49
pragma Align_members - set the maximum	boundary for structure-member alignment .....	268
array	bounds .....	79
	break or goto from the for-loop body .....	289
	break statement .....	104, 107, 109
operator functions versus	built-in operators .....	201
<b>C</b>		
differences between ANSI	C and C++ .....	1
lexical elements of	C and C++ .....	15
preprocessing	C and C++ program text .....	33
Standard	C for-loop construct .....	279
introduction of C++ features into	C programs .....	251
	C programs with some object orientation .....	135
macros for converting	C to C++ .....	257
migrating from	C to C++ .....	251
class objects in	C versus C++ .....	130
extern	"C" .....	71
problems with the	C++ global namespace .....	215
extern	"C++" .....	71, 255
using	C-style casts for conversion functions .....	183
	C/C++ compatibility issues .....	251
pragma calling_convention - allow High C/C++	call functions not compiled with High C/C++ .....	273
modules to	call mechanism .....	145
virtual	call using positional parameter notation .....	261
function	_CALLEE_POPS_STACK .....	273
calling-convention attribute	calling convention .....	273
Microsoft Pascal	calling-convention attribute _CALLEE_POPS_STACK ..	273
	calling-convention attribute _REVERSE_PARMS .....	273
	calling-convention attribute _SAVE_REGS .....	273
specifying the	calling-convention attribute with function declaration .....	273
pragma	Calling_convention - allow High C/C++ modules to call	
	functions not compiled with High C/C++ .....	273
overhead of function	calls .....	293
recursive	calls .....	285
	calls to function yield() .....	285, 290
High C/C++ extensions:	case ranges in switch statements .....	16, 103, 259
	case values .....	79
	case_label_statement .....	102
function-style	cast .....	255

explicit conversions by	cast_expression .....	90
run-time checks for validity of type	casting .....	97
type	casting .....	249, 250
	casting .....	96, 239, 247
	casting away const .....	248
	casting away volatile .....	248
	casting function addresses .....	266
	casts .....	271
syntactic ambiguity of declarations versus		
type	casts .....	243
situations in which static_cast	casts are valid .....	248
using C-style	casts for conversion functions .....	183
	casts implemented as templates .....	247
	casts of pointer types .....	271
exception	catch points .....	226
	catch statement syntax .....	227
the throw, try, and	catch statements .....	225, 226
the	catch(...) statement .....	230
	catching exceptions .....	225
High C/C++ extensions: keyword	_CC .....	19, 273
array of	char .....	31
data type	char .....	56
data type signed	char .....	56
data type unsigned	char .....	56
data type	char [] .....	31
	char_const .....	30
newline	character .....	16
line splice	character \ .....	33
backslash	character \ in escape sequences .....	30
	character constants .....	29
C++	character set .....	15
wide characters for international	character sets .....	30
wide	characters for international character sets .....	30
High C/C++ extensions: significant	characters in identifiers .....	259
access of a public or protected name		
inherited from a base	class .....	157
access specified for a base	class .....	153
access to protected members of a base	class .....	156
complete type of a subobject of a polymorphic	class .....	239, 240
defining a member function of a template	class .....	212
inheritance of members from a base	class .....	139
instance of a	class .....	5
restricting access to data within a	class .....	123
static type of a subobject of a polymorphic	class .....	240
storage for simple members of a struct or	class .....	129
type object of a polymorphic	class .....	241

	class compared with struct and union ..	62, 123, 126, 128, 138
	class declarations and definitions .....	123
	class hierarchy .....	138
members of the base	class inherited as members .....	153
non-static	class member function .....	149
	class member functions .....	5
access to	class members .....	5, 128
inaccessible	class members .....	151
scope of	class members .....	126
	class object .....	5
	class objects in C versus C++ .....	130
	class structure .....	5
	class templates .....	205
accessing members of a	class that are not public .....	159
implicit conversions from a derived	class to a base class .....	239
conversions from a base	class to a derived class .....	239
declaring a	class to be a base class .....	138
using dynamic_cast to make conversions from		
base	class to pointer class .....	242
	class type .....	128
default access rules for	class versus struct .....	125
declaring a	class within a block .....	127
	class-specific delete operator .....	91
	class-specific new operator .....	92
	class_definition .....	63, 125
	class_head .....	124
	class_key .....	124
	class_name .....	124, 175
abstract	classes .....	14, 146
accessible base	classes .....	86
C++	classes .....	123
derived and base	classes .....	137
derived	classes .....	137, 140
friend	classes .....	160
local	classes .....	127
multiple access through virtual base	classes .....	158
nested	classes .....	127
virtual base	classes .....	146
virtual base	classes and constructors .....	172
scope and access of local	classes and members .....	127
	classes as namespaces .....	223
using nested	classes with Incremental C++ .....	257
intermediate exception catching and	clean-up .....	229, 230
rethrowing an exception after	clean-up .....	229, 230
additional	clean-up during unwinding .....	229

	clean-up of completely constructed heap objects .....	234
	clean-up of partially constructed and destroyed heap objects .....	233, 234
automatic	clean-up with destructors .....	228
programming language	CLU .....	279
conversion	code .....	271
recursion and	code clarity .....	283
defining macro	code sequences and symbolic constants .....	36
	comma operator , .....	81
	commas in numbers .....	259
	comment style <code>/**/</code> .....	17
	comment style <code>//</code> .....	17, 33
	comments .....	17
splicing <code>//</code> -style	comments .....	34
	communicating with other languages .....	71
direct	communication between error and error handler .....	225
	comparing operands of different types .....	83
	comparing pointers to constant expressions .....	83
	comparing pointers to data members .....	84
type and assignment	compatibility .....	270
parameter type	compatibility and conversion .....	117
C/C++	compatibility issues .....	251
	<code>compile_time_constant</code> .....	40
	compiler-generated assignment .....	188
	compiler-generated destructors .....	178
	compiler-generated functions .....	209
	compiling in ANSI mode .....	272, 276
using keyword <code>typeid</code> to find the	complete type .....	240
	complete type of a subobject of a polymorphic class .....	239, 240
	<code>complete_class_name</code> .....	125
	compound statements or blocks .....	101
	concatenation .....	42
	<code>cond_expr</code> .....	107
nested	conditional directives .....	38
	conditional expressions .....	78
throw statements as	conditional operands .....	236
	conditional preprocessor directive <code>#if</code> .....	39
	conditional preprocessor directives <code>#elif</code> and <code>#else</code> .....	39
	conditional preprocessor directives <code>#ifdef</code> and <code>#ifndef</code> .....	39
casting away	<code>const</code> .....	248
function qualified with	<code>const</code> .....	136
keyword	<code>const</code> .....	49
type qualifier	<code>const</code> .....	67
keywords	<code>const</code> and <code>volatile</code> .....	113, 271
assignment operators accepting	<code>const</code> arguments .....	188
static	<code>const</code> object .....	68

declaring a	const or volatile function .....	255
	const_cast .....	247
specifying the radix of an integer	constant .....	275
comparing pointers to	constant expressions .....	83
	constant expressions .....	78
	constant expressions as template arguments .....	207
character	constants .....	29
decimal	constants .....	26
defining macro code sequences and symbolic	constants .....	36
floating-point	constants .....	27
function	constants .....	264
hexadecimal	constants .....	28
multi-character	constants .....	30
octal	constants .....	28
iterator syntax and	constraints .....	288
	constraints on using named parameters .....	262
clean-up of partially	constructed and destroyed heap objects .....	233, 234
clean-up of completely	constructed heap objects .....	234
	constructing an object .....	166
conversion by	constructor .....	179, 183
default	constructor .....	166
public copy	constructor .....	175
semantics of a generated copy	constructor .....	175
suitable	constructor .....	165
topmost	constructor .....	172
	constructor initializers .....	170
arguments to	constructors .....	92, 166
	constructors .....	143, 164, 165
copy	constructors .....	75, 164, 174
default	constructors .....	7
virtual base classes and	constructors .....	172
void return type for	constructors .....	256
	constructors and virtual functions .....	167
	constructors as non-static members .....	165
	control access to data members .....	152
access	control for declarations .....	153
flow of	control in a for loop .....	107
flow of	control in a switch .....	104
access	control in C++ .....	151
	control lines .....	35
implicit pointer	conversion .....	84
integral	conversion .....	194
multi-level pointer	conversion .....	248
numeric base	conversion .....	29
parameter type compatibility and	conversion .....	117
	conversion by constructor .....	179, 183



	conversion code .....	271
	conversion from a more precise type to a less precise type .....	93
	conversion functions .....	164, 180
using C-style casts for	conversion functions .....	183
arithmetic	conversions .....	94
automatic type	conversions .....	93
dangerous type	conversions .....	248
explicit and implicit pointer	conversions .....	97
expressions and	conversions .....	77
floating-point	conversions .....	93
implicit type	conversions .....	93, 97
integer	conversions .....	94
pointer	conversions .....	95
raw type	conversions .....	250
standard	conversions .....	194
user-defined	conversions .....	179
explicit	conversions by casting .....	97
implicit	conversions from a base class to a derived class .....	239
using dynamic_cast to make	conversions from a derived class to a base class .....	239
bitwise	conversions from base class to pointer class .....	242
destruction of the parameter	copies of the type object .....	241
public	copy .....	176
semantics of a generated	copy .....	230
parameter	copy constructor .....	175
predefined macro	copy constructor .....	175
High C/C++ extensions: predefined macro	copy constructors .....	75, 164, 174
	copy of a thrown expression .....	228
	__cplusplus .....	42
	__cplusplus .....	42
	cv_qualifier_list .....	49, 111

## D

automatic	data .....	51
declaring a	data item extern .....	51
allocating a	data item to a register .....	51
initializer of a static	data member .....	132
comparing pointers to	data members .....	84
control access to	data members .....	152
tree	data structures .....	284
function return	data type .....	112
	data type char .....	56
	data type char [] .....	31
	data type double .....	56
	data type float .....	56
	data type int .....	56

	data type long double .....	56
	data type long int .....	56
High C/C++ extensions:	data type long long int .....	56, 57
	data type short int .....	56
	data type signed char .....	56
default	data type signed int .....	56
	data type unsigned char .....	56
	data type void .....	56
	data type wchar_t .....	30
	data type wchar_t [] .....	31
arithmetic	data types .....	56
	data types .....	52
floating	data types .....	56
integral	data types .....	56
signed and unsigned	data types .....	56
size of	data types .....	57
order and	data types of function parameters .....	117
restricting access to	data within a class .....	123
predefined macro	__DATE__ .....	42
exception handlers as	dead code .....	236
	deallocating dynamic memory with operator delete .....	177
	decimal constants .....	26
	decl_specifier .....	48
!	declaration .....	291
bit-field	declaration .....	67, 130
	declaration .....	205
expression statement versus	declaration .....	101
specifying the calling-convention attribute		
with function	declaration .....	273
operator functions:	declaration and overloading .....	198
overriding the global	declaration of a name .....	221
pure	declaration of a virtual function .....	146
	declaration versus definition .....	47
access control for	declarations .....	153
	declarations .....	48
High C/C++ extensions: intermixing statements		
and	declarations .....	99
parameter	declarations .....	288
scope of template	declarations .....	207
class	declarations and definitions .....	123
	declarations and definitions .....	47
template	declarations and definitions .....	210
	declarations and precedence .....	70
	declarations in switch statements .....	103
	declarations of enum types .....	275
multiple	declarations using function templates .....	210

syntactic ambiguity of	declarations versus type casts .....	243
function	declarator .....	48, 52, 70, 271
overloaded name or operator	declarator .....	111
	declared friend .....	161
	declared_name .....	48
	declaring a class to be a base class .....	138
	declaring a class within a block .....	127
	declaring a const or volatile function .....	255
	declaring a data item extern .....	51
	declaring a function extern .....	51
	declaring a static function .....	51
	declaring arrays .....	61
	declaring structure members .....	63
High C/C++ extensions: keyword	__declspec .....	19
preprocessor directives	#define and #undef .....	36
	#define Incremental C++ macros .....	258
	defining a member function of a template class .....	212
	defining macro code sequences and symbolic constants ....	36
	defining names declared in a namespace .....	217
	defining structures .....	63
	defining types with typedef .....	69
declaration versus	definition .....	47
function	definition .....	113
iterator	definition .....	280
class declarations and	definitions .....	123
declarations and	definitions .....	47
template declarations and	definitions .....	210
deallocating dynamic memory with operator	delete .....	177
operator	delete .....	24, 90, 164, 184
class-specific	delete operator .....	91
	deleting arrays .....	90
automatic pointer	dereference .....	135
	dereference operator * .....	23, 87
postfix	dereference operator -> .....	203
	derived and base classes .....	137
conversions from a base class to a	derived class .....	239
implicit conversions from a	derived class to a base class .....	239
	derived classes .....	137, 140
	derived types .....	59
clean-up of partially constructed and	destroyed heap objects .....	233, 234
	destroyed object defined .....	229
order of	destruction .....	179
	destruction of the parameter copy .....	230
public	destructor .....	178
automatic clean-up with	destructors .....	228
compiler-generated	destructors .....	178

	destructors .....	7, 164, 176
	digit(s) .....	18, 27
	distance attributes .....	269, 271
	distance-attributed type .....	270
	division operator / .....	82
multiplication,	division, and modulus .....	82
data type	double .....	56
data type long	double .....	56
High C/C++ extensions: keyword	_Dpascal .....	19
inheritance and	duplicate member names .....	155
	duplicate name declarations with using .....	219
deallocating	dynamic memory with operator delete .....	177
allocating	dynamic memory with operator new .....	91
	dynamic_cast .....	247, 249
	dynamic_cast and reference types .....	246
using	dynamic_cast to make conversions from base class to pointer	
	class .....	242
using	dynamic_cast with pointer types .....	246

## E

conditional preprocessor directives	#elif and #else .....	39
	elif_line .....	38
The Annotated C++ Reference Manual by Margaret Ellis and Bjarne Stroustrup .....		1
conditional preprocessor directives #elif and	#else .....	39
	else_line .....	38
	encapsulating library names in a library namespace .....	215
	encapsulating names in your own code in a namespace ...	216
	encapsulation .....	5
initializing static	entities .....	73
type	enum .....	58
declarations of	enum types .....	275
toggle Long_enums - map type	enum variables to an int-sized area of memory .....	275
	enum_definition .....	58
	enumeration type .....	275
High C/C++ extensions: long and short	enumeration types .....	59, 275
	enumerators .....	58
initial values for	enumerators .....	58, 79
function parent and	environment .....	263
relational operator ==	(equal to) .....	83
template type	equivalence .....	207
throwing an exception on	error .....	225
preprocessor directives	#error and #print .....	40
direct communication between error and	error handler .....	225
problems with conventional	error handling .....	225

detecting	errors in code .....	225
	escape sequences .....	30
short-circuit	evaluation of logical AND and OR .....	85
parameter	evaluation order .....	119
expression	evaluation side effects .....	100
	exact-match function selection .....	209
C++	example .....	3
rethrowing an	exception after clean-up .....	229, 230
how an	exception becomes current .....	231
	exception catch points .....	226
intermediate	exception catching and clean-up .....	229, 230
what an	exception handler can catch .....	227, 236
	exception handlers .....	226
jumps with try blocks and	exception handlers .....	236
	exception handlers as dead code .....	236
how	exception handlers catch exceptions .....	227
	exception handling during unwinding .....	232
how	exception handling works .....	225
what happens when an	exception is thrown .....	227
throwing an	exception on error .....	225
function	exception specifications .....	234
the	exception-handler argument .....	227
C++	exception-handling .....	225
	exception-handling run time .....	227
active	exceptions .....	227, 231
catching	exceptions .....	225
how exception handlers catch	exceptions .....	227
miscellaneous rules for handling	exceptions .....	236
stack discipline of active	exceptions .....	231
stacked and nested	exceptions .....	231
bitwise	exclusive OR operator ^ .....	85
	exiting a loop without the use of goto .....	109
	Exponent .....	27
elements of an	expression .....	77
	expression .....	102, 106
parameter copy of a thrown	expression .....	228
type of an assignment	expression .....	80
using the throw statement without an	expression .....	230
	expression evaluation side effects .....	100
	expression statement versus declaration .....	101
array	expressions .....	62
comparing pointers to constant	expressions .....	83
conditional	expressions .....	78
constant	expressions .....	78
operands in	expressions .....	21
reference variables used in	expressions .....	60

side effects caused by	expressions .....	78
	expressions and conversions .....	77
	expressions as statements .....	100
constant	expressions as template arguments .....	207
	expressions passed to an iterator .....	288
High C/C++	extensions .....	259
High C/C++	extensions: arbitrary bases for numbers .....	29, 275
High C/C++	extensions: case ranges in switch statements ..	16, 103, 259
High C/C++	extensions: data type long long int .....	56, 57
High C/C++	extensions: goto statements in nested functions .....	110
High C/C++	extensions: initializing automatic aggregates .....	260
High C/C++	extensions: intermixing statements and declarations .....	99
High C/C++	extensions: intrinsics .....	268
High C/C++	extensions: keyword <code>__declspec</code> .....	19
High C/C++	extensions: keyword <code>_CC</code> .....	19, 273
High C/C++	extensions: keyword <code>_Dpascal</code> .....	19
High C/C++	extensions: keywords <code>_Packed</code> and <code>_Unpacked</code> ..	19, 63, 268
High C/C++	extensions: long and short enumeration types .....	59, 275
High C/C++	extensions: named parameter association .....	261, 289
High C/C++	extensions: near and far objects and pointers .....	268
High C/C++	extensions: nested functions .....	113
High C/C++	extensions: nested functions and full-function variables ..	263
High C/C++	extensions: packed and unpacked structures .....	268
High C/C++	extensions: predefined macro <code>__CPLUSPLUS__</code> .....	42
High C/C++	extensions: predefined macro <code>__HIGHC__</code> .....	42, 272
High C/C++	extensions: reserved word <code>pragma</code> .....	260
High C/C++	extensions: significant characters in identifiers .....	259
High C/C++	extensions: type qualifier <code>_Alias</code> .....	19, 69
High C/C++	extensions: type qualifier <code>_Far</code> .....	19, 269
High C/C++	extensions: type qualifier <code>_Near</code> .....	19, 269
High C/C++	extensions: type qualifier <code>_Noalias</code> .....	19, 69
High C/C++	extensions: underscores in numbers .....	27, 88, 259
declaring a data item	extern .....	51
declaring a function	extern .....	51
	extern "C" .....	71
	extern "C++" .....	71, 255
	extern "FORTRAN" .....	71
	extern "Pascal" .....	71
storage categories auto,	extern, register, and static .....	50
namespaces in	external mangled names .....	216, 219
	external storage .....	51
namespaces, friends, and	externs .....	222

## F

High C/C++ extensions: type qualifier	<code>_Far</code> .....	19, 269
---------------------------------------	-------------------------	---------

High C/C++ extensions: near and	far objects and pointers .....	268
	far pointers .....	270
bit	fields .....	66
	file scope .....	50
predefined macro	__FILE__ .....	42
	filename .....	40
High C/C++ include	files .....	253
data type	float .....	56
	float_suffix .....	27
	floating data types .....	56
	floating-point constants .....	27
	floating-point conversions .....	93
	flow of control in a for loop .....	107
	flow of control in a switch .....	104
flow of control in a	for loop .....	107
iterator-driven	for loop .....	282
	for loop body .....	106, 280, 284, 288, 291
Standard C	for-loop construct .....	279
	for loop index .....	107
	for loops .....	279, 283, 284, 285, 288, 289, 293
iterator-driven	for loops .....	279
semantics of iterators and	for loops .....	290
break or goto from the	for-loop body .....	289
	for_init .....	106
	foreign-language symbols .....	32
	formal and actual parameters .....	116
	formal parameter list .....	288
function yield()	formal-parameter list .....	289
extern	"FORTRAN" .....	71
function	frame .....	263
overloaded name or operator declared	friend .....	161
property of being a	friend .....	160
	friend classes .....	160
	friend functions .....	13
inline	friend functions .....	161
	friends .....	159
template	friends .....	212
namespaces,	friends, and externs .....	222
! following parameter list for	full-function or nested-function values .....	264
sizeof a	full-function type .....	266
function address versus	full-function value .....	264
High C/C++ extensions: nested functions and	full-function variables .....	263
address of a nested	function .....	263
address of an overloaded	function .....	87
declaring a const or volatile	function .....	255
declaring a static	function .....	51

nested (non-global)	function .....	291
non-static class member	function .....	149
overriding a virtual	function .....	141
pointer to a	function .....	264
pure declaration of a virtual	function .....	146
semantics of a virtual	function .....	141
	function address versus full-function value .....	264
casting	function addresses .....	266
	function body .....	113
	function call using positional parameter notation .....	261
overhead of	function calls .....	293
	function collection in a namespace .....	220
	function constants .....	264
specifying the calling-convention attribute		
with	function declaration .....	273
	function declarator .....	111
	function definition .....	113
	function exception specifications .....	234
declaring a	function extern .....	51
	function frame .....	263
	function labels .....	266
	function main .....	114
static	function members .....	132
defining a member	function of a template class .....	212
addresses as	function parameters .....	118
arrays as	function parameters .....	118
	function parameters .....	116
order and data types of	function parameters .....	117
	function parent and environment .....	263
	function prototypes .....	113
	function qualified with const .....	136
	function return data type .....	112
exact-match	function selection .....	209
	function set_terminate() .....	232
	function set_unexpected() .....	235
	function templates .....	208
multiple declarations using	function templates .....	210
customizing the behavior of	function terminate() .....	232
	function terminate() .....	232, 235
pointer-to-member of a	function type .....	149
customizing the behavior of	function unexpected() .....	235
calls to	function yield() .....	285, 290
parameters to	function yield() .....	289
predefined	function yield() .....	279, 280, 281, 288
	function yield() formal-parameter list .....	289
postfix	function-call operator () .....	202



	function-match overload resolution .....	196
	function-returning-far-int .....	270, 271
	function-returning-int .....	271
	function-style cast .....	255
	function_definition .....	63
	function_expression .....	202
class member	functions .....	5
compiler-generated	functions .....	209
constructors and virtual	functions .....	167
conversion	functions .....	164, 180
friend	functions .....	13
	functions .....	111
High C/C++ extensions: goto statements in		
nested	functions .....	110
High C/C++ extensions: nested	functions .....	113
inline friend	functions .....	161
inline	functions .....	115
inline member	functions .....	132
member	functions .....	132
non-static member	functions .....	113, 134
operator	functions .....	12
pure virtual	functions .....	14, 128
qualified member	functions .....	136
recursive	functions .....	113
scope of member	functions .....	133
special C++ member	functions .....	163
static member	functions .....	133
template member	functions .....	211
using C-style casts for conversion	functions .....	183
using special keywords and	functions .....	253
virtual	functions .....	13, 141, 144
High C/C++ extensions: nested	functions and full-function variables .....	263
name overloading and operator	functions in C++ .....	191
	functions nested within iterators .....	285
pragma calling_convention - allow High C/C++		
modules to call	functions not compiled with High C/C++ .....	273
	functions on type objects .....	244
making	functions overloadable .....	255
	functions qualified with volatile .....	136
operator	functions versus built-in operators .....	201
virtual	functions versus virtual bases .....	146
operator	functions: declaration and overloading .....	198
	fundamental types .....	56

## G

semantics of a	generated copy constructor .....	175
	generated default assignment operator .....	188
overriding the	global declaration of a name .....	221
exiting a loop without the use of	goto .....	109
break or	goto from the for-loop body .....	289
	goto statement .....	110
High C/C++ extensions:	goto statements in nested functions .....	110
non-local	gotos .....	267
relational operator >=	(greater than or equal to) .....	83
relational operator >	(greater than) .....	83

## H

profile	hc.pro .....	269
	header file typeinfo.h .....	244, 245
clean-up of completely constructed	heap objects .....	234
clean-up of partially constructed and destroyed	heap objects .....	233, 234
inconsistency in the treatment of	heap objects .....	234
single-copy	heuristic .....	242
	hex_digit .....	29
	hexadecimal constants .....	28
	hidden argument this 13, 134, 135, 136, 141, 147, 165	
class	hierarchy .....	138
differences between	High C syntax and others .....	271
iterators in	High C/C++ .....	279
	High C/C++ extensions .....	259
	High C/C++ extensions: arbitrary bases for numbers 29, 275	
	High C/C++ extensions: case ranges in switch statements .....	16, 103, 259
	High C/C++ extensions: data type long long int .....	56, 57
	High C/C++ extensions: goto statements in nested functions .....	110
	High C/C++ extensions: initializing automatic aggregates 260	
	High C/C++ extensions: intermixing statements and declarations .....	99
	High C/C++ extensions: intrinsics .....	268
	High C/C++ extensions: keyword __declspec .....	19
	High C/C++ extensions: keyword _CC .....	19, 273
	High C/C++ extensions: keyword _Dpascal .....	19
	High C/C++ extensions: keywords _Packed and _Unpacked .....	19, 63, 268
	High C/C++ extensions: long and short enumeration types .....	59, 275

	High C/C++ extensions: named parameter association ...	261, 289
	High C/C++ extensions: near and far objects and pointers	268
	High C/C++ extensions: nested functions .....	113
	High C/C++ extensions: nested functions and full-function variables .....	263
	High C/C++ extensions: packed and unpacked structures	268
	High C/C++ extensions: predefined macro <code>__CPLUSPLUS__</code> .....	42
	High C/C++ extensions: predefined macro <code>__HIGHC__</code> ..	42, 272
	High C/C++ extensions: reserved word <code>pragma</code> .....	260
	High C/C++ extensions: significant characters in identifiers .....	259
	High C/C++ extensions: type qualifier <code>_Alias</code> .....	19, 69
	High C/C++ extensions: type qualifier <code>_Far</code> .....	19, 269
	High C/C++ extensions: type qualifier <code>_Near</code> .....	19, 269
	High C/C++ extensions: type qualifier <code>_Noalias</code> .....	19, 69
	High C/C++ extensions: underscores in numbers	27, 88, 259
	High C/C++ include files .....	253
	High C/C++ keywords .....	19
pragma <code>calling_convention</code> - allow	High C/C++ modules to call functions not compiled with High C/C++ .....	273
High C/C++ extensions: predefined macro	<code>__HIGHC__</code> .....	42, 272
<b>I</b>		
	attributes of an identifier .....	47
High C/C++ extensions: significant characters	in identifiers .....	259
	in identifiers .....	18
	reserved identifiers .....	18
conditional preprocessor directive	<code>#if</code> .....	39
	if statement .....	104
	if <code>_line</code> .....	38
conditional preprocessor directives	<code>#ifdef</code> and <code>#ifndef</code> .....	39
conditional preprocessor directives <code>#ifdef</code>	and <code>#ifndef</code> .....	39
	importing an entire namespace .....	220
	inaccessible class members .....	151
	<code>inc_expr</code> .....	107
High C/C++	include files .....	253
nested	includes .....	36
using	Incremental C++ .....	252
using nested classes with	Incremental C++ .....	257
	Incremental C++ alternate keywords .....	254
<code>#define</code>	Incremental C++ macros .....	258

converting an	Incremental C++ program to a full C++ program .....	257
moving from	Incremental C++ to full C++ .....	257
C++ features you cannot use in	incremental mode .....	257
accessing C++ features	incrementally .....	254
for loop	index .....	107
direct and	indirect bases .....	140
	inheritance .....	137, 140
multiple	inheritance .....	147
single versus multiple	inheritance .....	138
	inheritance and duplicate member names .....	155
	inheritance of members from a base class .....	139
members of the base class	inherited as members .....	153
access of a public or protected name	inherited from a base class .....	157
	init_declarator .....	48
	init_value .....	72
	initial values for enumerators .....	58, 79
	initialization .....	72
explicit	initialization in conditions .....	243
order of	initialization of bases and members .....	171
	initialization of static variables .....	254
C++ parenthesized	initializer .....	256
	initializer .....	72, 73, 91
	initializer of a static data member .....	132
constructor	initializers .....	170
scope of	initializers .....	73
	initializing a reference .....	61
	initializing arrays .....	72
High C/C++ extensions:	initializing automatic aggregates .....	260
	initializing objects with operator new .....	92
	initializing static entities .....	73
	initializing structure members .....	72
	inline friend functions .....	161
	inline functions .....	115
	inline member functions .....	132
	inlining as an optimization .....	115
	inlining versus macros .....	115
	inner blocks .....	49
iterator	input arguments .....	288
	instance of a class .....	5
overriding a template	instantiation .....	211
data type	int .....	56
data type long	int .....	56
data type short	int .....	56
default data type signed	int .....	56
High C/C++ extensions: data type long long	int .....	56, 57

toggle Long_enums - map type enum variables to an	int-sized area of memory .....	275
maximum and minimum values for an	integer .....	26
specifying the radix of an	integer constant .....	275
	integer conversions .....	94
	integral conversion .....	194
	integral data types .....	56
	integral promotion .....	94, 194
AMD and	Intel architectures .....	270, 271
	intermediate exception catching and clean-up .....	229, 230
High C/C++ extensions:	intermixing statements and declarations .....	99
wide characters for	internal linkage within an unnamed namespace .....	223
High C/C++ extensions:	international character sets .....	30
improved type checking at the iterator	intrinsics .....	268
	invocation .....	287
	invoking an iterator .....	288
	iterating over each element in a set .....	281
	iterating through the nodes of a binary tree .....	283
loop	iteration .....	279, 292
	iteration statements .....	105
expressions passed to an	iterator .....	288
invoking an	iterator .....	288
results yielded by the	iterator .....	280, 287
types yielded by an	iterator .....	289
yield parameter of an	iterator .....	291
	iterator definition .....	280
	iterator input arguments .....	288
improved type checking at the	iterator invocation .....	287
special	iterator syntax -> .....	280
	iterator syntax and constraints .....	288
	iterator-driven for loop .....	282
	iterator-driven for loops .....	279
what is an	iterator? .....	279
advantages and disadvantages of	iterators .....	292
arguments to	iterators .....	288
functions nested within	iterators .....	285
replacing macros with	iterators .....	285
semantics of	iterators .....	289
uses for	iterators .....	281
semantics of	iterators and for loops .....	290
	iterators in High C/C++ .....	279
how	iterators work .....	280

## J

jump statements .....	108
-----------------------	-----

jumps with try blocks and exception handlers ..... 236

## K

	Kanji support .....	274
High C/C++ extensions:	keyword <code>__declspec</code> .....	19
High C/C++ extensions:	keyword <code>_CC</code> .....	19, 273
High C/C++ extensions:	keyword <code>_Dpascal</code> .....	19
	keyword <code>const</code> .....	49
	keyword <code>private</code> .....	5
	keyword <code>protected</code> .....	5
	keyword <code>public</code> .....	5
using	keyword <code>typeid</code> to find the complete type .....	240
C++	keywords .....	18
High C/C++	keywords .....	19
Incremental C++ alternate	keywords .....	254
High C/C++ extensions:	keywords <code>_Packed</code> and <code>_Unpacked</code> .....	19, 63, 268
using special	keywords and functions .....	253
	keywords <code>const</code> and <code>volatile</code> .....	113, 271
converting to	known and unknown types .....	181

## L

default	label .....	104
	labeled statements .....	100
function	labels .....	266
non-local	labels .....	266
scope of statement	labels .....	100
programming	language Ada .....	261
programming	language CLU .....	279
programming	language Pascal .....	263
	language_name .....	71
communicating with other	languages .....	71
	late binding .....	13
	left- and right- associative operators .....	272
	leftshift operator <code>&lt;&lt;</code> .....	84
bit-field	lengths .....	79
relational operator <code>&lt;=</code>	(less than or equal to) .....	83
relational operator <code>&lt;</code>	(less than) .....	83
	letters .....	18
	lexical elements of C and C++ .....	15
	lexical syntax for a number .....	276
encapsulating	library names in a library namespace .....	215
preprocessor directive	<code>#line</code> .....	40
	line splice character <code>\</code> .....	33
	line splicing .....	33

predefined macro	<code>__LINE__</code>	42
control	lines	35
static	link	114
	linkage specifications	71
using static	links in up-level referencing	263
	local classes	127
scope and access of	local classes and members	127
	logical AND and OR	85
	logical AND operator <code>&amp;&amp;</code>	85
	logical NOT operator <code>!</code>	23, 88
	logical OR operator <code>  </code>	85
	<code>logical_expression</code>	78
High C/C++ extensions:	long and short enumeration types	59, 275
data type	long double	56
data type	long int	56
High C/C++ extensions: data type	long long int	56, 57
toggle	Long_enums - map type enum variables to an int-sized area of memory	275
	loop iteration	279, 292
	loop overhead	293
exiting a	loop without the use of <code>goto</code>	109
	lvalues	77

## M

	machine registers	293
ANSI C offsetof	macro	149
predefined	macro <code>__cplusplus</code>	42
High C/C++ extensions: predefined	macro <code>__cplusplus</code>	42
predefined	macro <code>__DATE__</code>	42
predefined	macro <code>__FILE__</code>	42
High C/C++ extensions: predefined	macro <code>__HIGHC__</code>	42, 272
predefined	macro <code>__LINE__</code>	42
predefined	macro <code>__STDC__</code>	42
predefined	macro <code>__TIME__</code>	42
defining	macro code sequences and symbolic constants	36
predefined	macro names	42
	<code>macro_name</code>	36
#define Incremental C++	macros	258
inlining versus	macros	115
va_arg	macros	289
	macros for converting C to C++	257
	macros <code>va_start()</code> , <code>va_arg()</code> , and <code>va_end()</code>	120
replacing	macros with iterators	285
function	main	114
namespaces in external	mangled names	216, 219

	Mantissa .....	27
argument type	matching .....	194
	maximum and minimum values for an integer .....	26
pragma Align_members - set the	maximum boundary for structure-member alignment .....	268
initializer of a static data	member .....	132
offset of a	member .....	149
non-static class	member function .....	149
defining a	member function of a template class .....	212
class	member functions .....	5
inline	member functions .....	132
	member functions .....	132
non-static	member functions .....	113, 134
qualified	member functions .....	136
scope of	member functions .....	133
special C++	member functions .....	163
static	member functions .....	133
template	member functions .....	211
inheritance and duplicate	member names .....	155
structure	member or element .....	62
	member type .....	128
	member_declaration .....	63, 124, 126
	member_declarator .....	124, 128
	member_declarator_list .....	124
	member_list .....	65, 124
access to class	members .....	5, 128
comparing pointers to data	members .....	84
constructors as non-static	members .....	165
control access to data	members .....	152
controlling the alignment of structure	members .....	63
declaring structure	members .....	63
inaccessible class	members .....	151
initializing structure	members .....	72
order of initialization of bases and	members .....	171
pointers to	members .....	148
scope and access of local classes and	members .....	127
scope of class	members .....	126
simple	members .....	128
static bit-field	members .....	132
static function	members .....	132
static	members .....	131
typedef	members .....	131
static	members and variables in a template .....	212
inheritance of	members from a base class .....	139
access to protected	members of a base class .....	156
accessing	members of a class that are not public .....	159
storage for simple	members of a struct or class .....	129



storage for simple	members of a union .....	129
toggle Long_enums - map type enum variables to an int-sized area of deallocating dynamic allocating dynamic	members of the base class inherited as members .....	153
	memory .....	275
	memory with operator delete .....	177
	memory with operator new .....	91
	methods .....	132
	Microsoft C .....	271
	Microsoft Pascal calling convention .....	273
	migrating from C to C++ .....	251
	minimum values for an integer .....	26
maximum and unary	minus operator - .....	23, 88
	mixed-model programming .....	273
	modifying types .....	67
pragma calling_convention - allow High C/C++ multiplication, division, and	modules to call functions not compiled with High C/C++ .....	273
	modulus .....	82
	multi-character constants .....	30
implementing a	multi-dimensional array .....	202
	multi-dimensional arrays .....	62
	multiple access through virtual base classes .....	158
	multiple declarations using function templates .....	210
	multiple inheritance .....	147
single versus	multiple inheritance .....	138
	multiplication operator * .....	82
	multiplication, division, and modulus .....	82

## N

	name .....	49
scope of a duplicate	name .....	50
access of a public or protected overloaded	name declarations with using .....	219
	name inherited from a base class .....	157
	name or operator declared friend .....	161
	name overloading and access .....	156
	name overloading and operator functions in C++ .....	191
blocks and High C/C++ extensions:	name scoping .....	49
	named parameter association .....	261, 289
	named parameter association suffix .....	262
	named parameter syntax .....	261
constraints on using inheritance and duplicate member predefined macro uniqueness of uniqueness of template	named parameters .....	262
	names .....	155
	names .....	42
	names .....	50
	names .....	207
defining using	names declared in a namespace .....	217
	names declared in a namespace .....	217

names declared in multiple namespaces .....	219
scope of	
explicit qualification and	
names in a for statement .....	106
names introduced by the using directive .....	222
names owned by a namespace .....	216, 219
names visible with typedefs .....	130
names within structures .....	257
namespace .....	216
namespace .....	217
namespace .....	215
namespace .....	216
namespace .....	220
namespace .....	216
namespace .....	220
namespace .....	223
namespace .....	216, 219
namespace .....	215
namespace .....	217
namespace aliasing definitions .....	217
namespace definition .....	216
namespaces .....	223
namespaces .....	223
namespaces .....	219
namespaces .....	215
namespaces .....	224
namespaces .....	222
namespaces .....	219
namespaces .....	215
namespaces in external mangled names .....	216, 219
namespaces, friends, and externs .....	222
_Near .....	19, 269
near and far objects and pointers .....	268
nested (non-global) function .....	291
nested classes .....	127
nested classes with Incremental C++ .....	257
nested conditional directives .....	38
nested exceptions .....	231
nested function .....	263
nested functions .....	110
nested functions .....	113
nested functions and full-function variables .....	263
nested includes .....	36
nested switch statements .....	104
nested within iterators .....	285
nested-function values .....	264
new .....	91
making type	
typedef	
continuing a	
defining names declared in a	
encapsulating library names in a library	
encapsulating names in your own code in a	
function collection in a	
how to specify a	
importing an entire	
internal linkage within an unnamed	
names owned by a	
problems with the C++ global	
using names declared in a	
the	
classes as	
current status of	
names declared in multiple	
templates and	
unnamed	
using declarations inside	
when to use	
High C/C++ extensions: type qualifier	
High C/C++ extensions:	
using	
stacked and	
address of a	
High C/C++ extensions: goto statements in	
High C/C++ extensions:	
High C/C++ extensions:	
functions	
! following parameter list for full-function	
or	
allocating dynamic memory with operator	

arguments to operator	new	91, 92, 256
initializing objects with operator	new	92
operator	new	7, 24, 91, 164, 184
class-specific	new operator	92
	newline character	16
High C/C++ extensions: type qualifier	_Noalias	19, 69
iterating through the	nodes of a binary tree	283
	non-local gotos	267
	non-local labels	266
	non-static class member function	149
	non-static member functions	113, 134
constructors as	non-static members	165
	nonzero_digit	26
relational operator !=	(not equal to)	83
logical	NOT operator !	23, 88
bitwise	NOT operator ~	23, 88
	null directive	35
	null pointer	95
default segment	number	268
lexical syntax for a	number	276
specifying the radix of a	number	29
variable	number of arguments	289
ANSI preprocessor	numbers	276
binary	numbers	275
commas in	numbers	259
High C/C++ extensions: arbitrary bases for	numbers	29, 275
High C/C++ extensions: underscores in	numbers	27, 88, 259
	numeric base conversion	29

## O

allocating storage for an	object	91
class	object	5
constructing an	object	166
copies of the type	object	241
static const	object	68
using the type	object	242
widened	object	94
types a type	object can represent	245
destroyed	object defined	229
type	object of a polymorphic class	241
allocating an	object of array type	91
C programs with some	object orientation	135
	object-oriented programming	123, 137
C++	objects	123
functions on type	objects	244

	objects .....	77
type	objects .....	240
High C/C++ extensions: near and far	objects and pointers .....	268
class	objects in C versus C++ .....	130
initializing	objects with operator new .....	92
	octal constants .....	28
	octal_digit .....	28
	offset of a member .....	149
ANSI C	offsetof macro .....	149
pointers as	operands .....	82
	operands in expressions .....	21
comparing	operands of different types .....	83
class-specific delete	operator .....	91
class-specific new	operator .....	92
generated default assignment	operator .....	188
sizeof	operator .....	57, 90
special default assignment	operator .....	188
topmost assignment	operator .....	188
logical NOT	operator ! .....	23, 88
relational	operator != (not equal to) .....	83
remainder	operator % .....	82
address	operator & .....	87
bitwise AND	operator & .....	85
logical AND	operator && .....	85
postfix function-call	operator () .....	202
dereference	operator * .....	23, 87
multiplication	operator * .....	82
addition	operator + .....	81
unary plus	operator + .....	23, 87
post-increment	operator ++ .....	23, 89
pre-increment	operator ++ .....	23, 88
comma	operator , .....	81
subtraction	operator - .....	81
unary minus	operator - .....	23, 88
post-decrement	operator -- .....	24, 89
pre-decrement	operator -- .....	24, 88
postfix dereference	operator -> .....	203
pointer-to-member	operator ->* .....	86
division	operator / .....	82
scope resolution	operator :: .....	49, 55, 73, 130, 145, 147
relational	operator < (less than) .....	83
leftshift	operator << .....	84
relational	operator <= (less than or equal to) .....	83
assignment	operator = .....	79, 186
binary	operator = .....	201
	operator =(0) .....	164

relational	operator == (equal to) .....	83
relational	operator > (greater than) .....	83
relational	operator >= (greater than or equal to) .....	83
right shift	operator >> .....	84
ternary	operator ?: .....	78
postfix subscript	operator [] .....	202
bitwise exclusive OR	operator ^ .....	85
bitwise OR	operator   .....	85
logical OR	operator    .....	85
bitwise NOT	operator ~ .....	23, 88
overloaded name or	operator declared friend .....	161
unary preprocessor	operator defined .....	40
deallocating dynamic memory with	operator delete .....	177
	operator delete .....	24, 90, 164, 184
	operator functions .....	12
name overloading and	operator functions in C++ .....	191
	operator functions versus built-in operators .....	201
	operator functions: declaration and overloading .....	198
allocating dynamic memory with	operator new .....	91
arguments to	operator new .....	91, 92, 256
initializing objects with	operator new .....	92
	operator new .....	7, 24, 91, 164, 184
	operator sizeof .....	24
additive	operators .....	81
associativity of	operators .....	20
binary	operators .....	24, 82, 201
equality	operators .....	83
left- and right- associative	operators .....	272
operator functions versus built-in	operators .....	201
	operators .....	79
pointer-to-member	operators .....	86
precedence of relational	operators .....	83
prefix unary	operators .....	201
relational	operators .....	83
shift	operators .....	84
subscript	operators .....	62
ternary	operators .....	26
unary	operators .....	20, 23, 87
arguments to	operators # and ## .....	42
uses of	operators # and ## .....	43
postfix unary	operators ++ and -- .....	201
pointer-to-member	operators .* and -> .....	86
assignment	operators accepting const arguments .....	188
ambiguities between user-defined	operators and pre-defined C++ operators .....	182
inlining as an	optimization .....	115
bitwise exclusive	OR operator ^ .....	85

bitwise	OR operator   .....	85
logical	OR operator    .....	85
parameter evaluation	order .....	119
	order and data types of function parameters .....	117
	order of destruction .....	179
	order of initialization of bases and members .....	171
	other_args .....	91
loop	overhead .....	293
	overhead of function calls .....	293
argument-match	overload resolution .....	192
function-match	overload resolution .....	196
	overload resolution .....	191
making functions	overloadable .....	255
address of an	overloaded function .....	87
	overloaded functions with the same argument types .....	218
	overloaded name or operator declared friend .....	161
operator functions: declaration and	overloading .....	198
name	overloading and access .....	156
name	overloading and operator functions in C++ .....	191
	overloading and scope .....	197
	overloading functions with the using declaration .....	218
access	override .....	153, 157
	overriding a template instantiation .....	211
	overriding a virtual function .....	141
	overriding the global declaration of a name .....	221

## P

Ada	packages .....	215
High C/C++ extensions: keywords	_Packed and _Unpacked .....	19, 63, 268
High C/C++ extensions:	packed and unpacked structures .....	268
High C/C++ extensions: named	parameter association .....	261, 289
named	parameter association suffix .....	262
destruction of the	parameter copy .....	230
	parameter copy of a thrown expression .....	228
	parameter declarations .....	288
	parameter evaluation order .....	119
formal	parameter list .....	288
prototype-form	parameter list .....	280
! following	parameter list for full-function or nested-function values .....	264
variable-length	parameter lists .....	119
function call using positional	parameter notation .....	261
yield	parameter of an iterator .....	291
named	parameter syntax .....	261
	parameter type compatibility and conversion .....	117
addresses as function	parameters .....	118

arrays as function	parameters .....	118
constraints on using named	parameters .....	262
default	parameters .....	121
formal and actual	parameters .....	116
function	parameters .....	116
order and data types of function	parameters .....	117
reference	parameters .....	118
	parameters argc and argv .....	114
passing	parameters by value and by reference .....	117
	parameters to function yield() .....	289
function	parent and environment .....	263
C++	parenthesized initializer .....	256
C++ constructs that introduce	parsing ambiguities .....	255
programming language	Pascal .....	263
extern	"Pascal" .....	71
	passing parameters by value and by reference .....	117
token	pasting .....	276
unary	plus operator + .....	23, 87
null	pointer .....	95
this	pointer .....	141
	pointer assignment .....	60
using dynamic_cast to make conversions from		
base class to	pointer class .....	242
implicit	pointer conversion .....	84
multi-level	pointer conversion .....	248
explicit and implicit	pointer conversions .....	97
	pointer conversions .....	95
automatic	pointer dereference .....	135
	pointer to a function .....	264
	pointer to void .....	95, 96
casts of	pointer types .....	271
using dynamic_cast with	pointer types .....	246
	pointer-to-far .....	271
	pointer-to-member of a function type .....	149
	pointer-to-member operator ->* .....	86
	pointer-to-member operators .....	86
	pointer-to-member operators .* and -> .....	86
	pointer-to-near .....	271
far	pointers .....	270
High C/C++ extensions: near and far objects		
and	pointers .....	268
implementing smart	pointers .....	203
	pointers .....	60
size of	pointers .....	97
subtracting	pointers .....	81
virtual base	pointers .....	147

virtual-function-table	pointers .....	145
	pointers as operands .....	82
comparing	pointers to constant expressions .....	83
comparing	pointers to data members .....	84
	pointers to members .....	148
side effects and sequence	points .....	89
complete type of a subobject of a	polymorphic class .....	239, 240
static type of a subobject of a	polymorphic class .....	240
type object of a	polymorphic class .....	241
	polymorphic typeid .....	241
	polymorphism .....	13, 145
function call using	positional parameter notation .....	261
	positional prefix .....	262
post-increment and	post-decrement .....	89
	post-decrement operator -- .....	24, 89
	post-increment and post-decrement .....	89
	post-increment operator ++ .....	23, 89
	postfix dereference operator -> .....	203
	postfix function-call operator () .....	202
	postfix subscript operator [] .....	202
	postfix unary operators ++ and -- .....	201
preprocessor directive	#pragma .....	41
High C/C++ extensions: reserved word	pragma .....	260
	pragma Align_members - set the maximum boundary for structure-member alignment .....	268
	pragma Calling_convention - allow High C/C++ modules to call functions not compiled with High C/C++ .....	273
pre-increment and	pre-decrement .....	88
	pre-decrement operator -- .....	24, 88
ambiguities between user-defined operators and	pre-defined C++ operators .....	182
	pre-increment and pre-decrement .....	88
	pre-increment operator ++ .....	23, 88
declarations and	precedence .....	70
	precedence of relational operators .....	83
	precedence of the throw expression .....	226
conversion from a more rounding and loss of	precise type to a less precise type .....	93
	precision .....	94
	predefined function yield() .....	279, 280, 281, 288
	predefined macro __cplusplus .....	42
High C/C++ extensions:	predefined macro __cplusplus .....	42
	predefined macro __DATE__ .....	42
	predefined macro __FILE__ .....	42
High C/C++ extensions:	predefined macro __HIGHC__ .....	42, 272
	predefined macro __LINE__ .....	42
	predefined macro __STDC__ .....	42



	predefined macro <code>__TIME__</code> .....	42
	predefined macro names .....	42
positional	prefix .....	262
	prefix unary operators .....	201
	preprocessing C and C++ program text .....	33
conditional	preprocessor directive <code>#if</code> .....	39
	preprocessor directive <code>#line</code> .....	40
	preprocessor directive <code>#pragma</code> .....	41
	preprocessor directives .....	35
	preprocessor directives <code>#assert</code> and <code>#unassert</code> .....	276
	preprocessor directives <code>#define</code> and <code>#undef</code> .....	36
conditional	preprocessor directives <code>#elif</code> and <code>#else</code> .....	39
	preprocessor directives <code>#error</code> and <code>#print</code> .....	40
conditional	preprocessor directives <code>#ifdef</code> and <code>#ifndef</code> .....	39
ANSI	preprocessor numbers .....	276
unary	preprocessor operator <code>defined</code> .....	40
preprocessor directives <code>#error</code> and	<code>#print</code> .....	40
keyword	<code>private</code> .....	5
public, protected, and	<code>private</code> access .....	151
access specifiers	<code>private</code> , <code>protected</code> , and <code>public</code> .....	64, 153
	profile <code>hc.pro</code> .....	269
abstractions in	program construction .....	279
preprocessing C and C++	program text .....	33
converting an Incremental C++	program to a full C++ program .....	257
mixed-model	programming .....	273
object-oriented	programming .....	123, 137
	programming language Ada .....	261
	programming language CLU .....	279
	programming language Pascal .....	263
introduction of C++ features into C	programs .....	251
C	programs with some object orientation .....	135
integral	<code>promotion</code> .....	94, 194
keyword	<code>protected</code> .....	5
access to	<code>protected</code> members of a base class .....	156
access of a public or	<code>protected</code> name inherited from a base class .....	157
public,	<code>protected</code> , and <code>private</code> access .....	151
access specifiers <code>private</code> ,	<code>protected</code> , and <code>public</code> .....	64, 153
ANSI	<code>prototype</code> syntax .....	288
	<code>prototype-form</code> parameter list .....	280
function	<code>prototypes</code> .....	113
	<code>ptr_operator</code> .....	49, 60, 70
access specifiers <code>private</code> , <code>protected</code> , and	<code>public</code> .....	64, 153
accessing members of a class that are not	<code>public</code> .....	159
keyword	<code>public</code> .....	5
	<code>public</code> access implied by <code>struct</code> or <code>union</code> .....	153
	<code>public</code> copy constructor .....	175

	public destructor .....	178
access of a	public or protected name inherited from a base class .....	157
	public, protected, and private access .....	151
	punctuator ",'" .....	16
	punctuator ".." .....	16
	punctuator "... " .....	16
	punctuator "':" .....	16
	punctuator "':" .....	16
	punctuators ( and ) .....	16
	punctuators [ and ] .....	16
	punctuators { and } .....	16
	pure declaration of a virtual function .....	146
	pure virtual functions .....	14, 128
	pure_specifier .....	125

## Q

	qualified member functions .....	136
function	qualified with const .....	136
functions	qualified with volatile .....	136
	qualified_class_name .....	125
register	qualifier .....	87
type	qualifiers .....	67

## R

specifying the	radix of a number .....	29
specifying the	radix of an integer constant .....	275
High C/C++ extensions: case	ranges in switch statements .....	16, 103, 259
	raw type conversions .....	250
	recursion and code clarity .....	283
	recursive calls .....	285
	recursive computation .....	282
	recursive functions .....	113
	recursive tree walk .....	283, 284
initializing a	reference .....	61
passing parameters by value and by	reference .....	117
The Annotated C++	Reference Manual by Margaret Ellis and Bjarne Stroustrup	1
	reference parameters .....	118
dynamic_cast and	reference types .....	246
	reference variables used in expressions .....	60
	reference_to_previous_class_or_enum_definition .....	54
	references .....	60
up-level	referencing .....	263
allocating a data item to a	register .....	51
	register qualifier .....	87

	register variables .....	267
storage categories auto, extern,	register, and static .....	50
machine	registers .....	293
	reinterpret_cast .....	247, 250
	relational operator != (not equal to) .....	83
	relational operator < (less than) .....	83
	relational operator <= (less than or equal to) .....	83
	relational operator == (equal to) .....	83
	relational operator > (greater than) .....	83
	relational operator >= (greater than or equal to) .....	83
precedence of	relational operators .....	83
	relational operators .....	83
	remainder operator % .....	82
	reserved identifiers .....	18
High C/C++ extensions:	reserved word pragma .....	260
C++	reserved words .....	1
argument-match overload	resolution .....	192
function-match overload	resolution .....	196
overload	resolution .....	191
scope	resolution operator :: .....	49, 55, 73, 130, 145, 147
	restricting access to data within a class .....	123
	results yielded by the iterator .....	280, 287
	rethrowing an exception after clean-up .....	229, 230
function	return data type .....	112
	return statement .....	109
void	return type for constructors .....	256
attributed	return types .....	270
calling-convention attribute	_REVERSE_PARMS .....	273
	right shift operator >> .....	84
left- and	right- associative operators .....	272
	rounding and loss of precision .....	94
access	rules .....	164
default access	rules for class versus struct .....	125
miscellaneous	rules for handling exceptions .....	236
exception-handling	run time .....	227
	run-time checks for validity of type casting .....	249, 250
	run-time type information .....	239
	rvalues .....	78

## S

calling-convention attribute	_SAVE_REGS .....	273
	scalar types .....	59
alternate	scan tables .....	274
file	scope .....	50
overloading and	scope .....	197

	scope access .....	55
	scope and access of local classes and members .....	127
	scope of a name .....	50
	scope of class members .....	126
	scope of initializers .....	73
	scope of member functions .....	133
	scope of names in a for statement .....	106
	scope of statement labels .....	100
	scope of template declarations .....	207
	scope resolution operator :: ...	49, 55, 73, 130, 145, 147
blocks and name	scoping .....	49
default	segment number .....	268
exact-match function	selection .....	209
	selection statement .....	102
	semantics of a generated copy constructor .....	175
	semantics of a virtual function .....	141
	semantics of iterators .....	289
	semantics of iterators and for loops .....	290
side effects and	sequence points .....	89
escape	sequences .....	30
defining macro code	sequences and symbolic constants .....	36
function	set_terminate() .....	232
function	set_unexpected() .....	235
High C/C++ extensions: long and	short enumeration types .....	59, 275
data type	short int .....	56
	short-circuit evaluation of logical AND and OR .....	85
expression evaluation	side effects .....	100
	side effects .....	119
	side effects and sequence points .....	89
	side effects caused by expressions .....	78
	signed and unsigned data types .....	56
data type	signed char .....	56
default data type	signed int .....	56
High C/C++ extensions:	significant characters in identifiers .....	259
	simple members .....	128
storage for	simple members of a struct or class .....	129
storage for	simple members of a union .....	129
	single versus multiple inheritance .....	138
	single-copy heuristic .....	242
	size of data types .....	57
	size of pointers .....	97
operator	sizeof .....	24
	sizeof a full-function type .....	266
	sizeof operator .....	57, 90
	sizeof value of unions .....	129
implementing	smart pointers .....	203

	source file analyze.c .....	266, 289
	source_text .....	38
blank	spaces .....	16
	special C++ member functions .....	163
	special default assignment operator .....	188
	special iterator syntax -> .....	280
using	special keywords and functions .....	253
sticky virtual	specifier .....	145
type	specifiers .....	54
access	specifiers private, protected, and public .....	64, 153
line	splice character \ .....	33
line	splicing .....	33
	splicing //-style comments .....	34
	stack discipline of active exceptions .....	231
	stacked and nested exceptions .....	231
ANSIX3J16	standard .....	1, 209
	Standard C for loop construct .....	279
	standard conversions .....	194
ANSIC++	standardization .....	1, 251
break	statement .....	104, 107, 109
continue	statement .....	105, 107, 109
do	statement .....	105
for	statement .....	106
goto	statement .....	110
if	statement .....	104
return	statement .....	109
scope of names in a for	statement .....	106
selection	statement .....	102
	statement .....	106, 107
switch	statement .....	102
the catch(...)	statement .....	230
while	statement .....	106
scope of	statement labels .....	100
expression	statement versus declaration .....	101
declarations in switch	statements .....	103
expressions as	statements .....	100
High C/C++ extensions: case ranges in switch	statements .....	16, 103, 259
iteration	statements .....	105
jump	statements .....	108
labeled	statements .....	100
nested switch	statements .....	104
	statements .....	99
High C/C++ extensions: intermixing	statements and declarations .....	99
High C/C++ extensions: goto	statements in nested functions .....	110
compound	statements or blocks .....	101

storage categories auto, extern, register, and	static .....	50
storage category	static .....	133
	static bit-field members .....	132
	static const object .....	68
initializer of a	static data member .....	132
initializing	static entities .....	73
declaring a	static function .....	51
	static function members .....	132
	static link .....	114
using	static links in up-level referencing .....	263
	static member functions .....	133
	static members .....	131
	static members and variables in a template .....	212
	static type of a subobject of a polymorphic class .....	240
initialization of	static variables .....	254
	static_cast .....	247
situations in which	static_cast casts are valid .....	248
predefined macro	__STDC__ .....	42
	sticky virtual specifier .....	145
external	storage .....	51
	storage allocation for direct bases .....	140
	storage categories auto, extern, register, and static .....	50
	storage category static .....	133
allocating	storage for an object .....	91
	storage for simple members of a struct or class .....	129
	storage for simple members of a union .....	129
	storage_class_specifier .....	52
	strings .....	31
The Annotated C++ Reference Manual by Margaret Ellis and Bjarne	Stroustrup .....	1
default access rules for class versus	struct .....	125
class compared with	struct and union .....	62, 123, 126, 128, 138
storage for simple members of a	struct or class .....	129
public access implied by	struct or union .....	153
	structs in C++ .....	1, 2
class	structure .....	5
	structure member or element .....	62
controlling the alignment of	structure members .....	63
declaring	structure members .....	63
initializing	structure members .....	72
pragma Align_members - set the maximum		
boundary for	structure-member alignment .....	268
defining	structures .....	63
High C/C++ extensions: packed and unpacked	structures .....	268
	structures .....	62

tree data	structures .....	284
typedef names within	structures .....	257
	structures and unions .....	62
comment	style <code>/**/</code> .....	17
comment	style <code>//</code> .....	17, 33
splicing	<code>//</code> -style comments .....	34
complete type of a	subobject of a polymorphic class .....	239, 240
static type of a	subobject of a polymorphic class .....	240
postfix	subscript operator <code>[]</code> .....	202
	subscript operators .....	62
array	subscripts .....	61
token	substitution .....	42
	subtracting pointers .....	81
addition and	subtraction .....	81
	subtraction operator <code>-</code> .....	81
named parameter association	suffix .....	262
Kanji	support .....	274
	<code>sw_statement</code> .....	102
flow of control in a	switch .....	104
	switch statement .....	102
declarations in	switch statements .....	103
High C/C++ extensions: case ranges in	switch statements .....	16, 103, 259
nested	switch statements .....	104
defining macro code sequences and	symbolic constants .....	36
foreign-language	symbols .....	32
	syntactic ambiguity of declarations versus type casts .....	243
ANSI prototype	syntax .....	288
catch statement	syntax .....	227
named parameter	syntax .....	261
try statement	syntax .....	226
using-directive	syntax .....	220
special iterator	syntax <code>-&gt;</code> .....	280
iterator	syntax and constraints .....	288
differences between High C	syntax and others .....	271
lexical	syntax for a number .....	276

## T

alternate scan	tables .....	274
virtual-function	tables .....	145, 169, 178
	tabs .....	16
static members and variables in a	template .....	212
constant expressions as	template arguments .....	207
	template arguments .....	206
defining a member function of a	template class .....	212
scope of	template declarations .....	207

	template declarations and definitions .....	210
	template friends .....	212
overriding a	template instantiation .....	211
	template member functions .....	211
uniqueness of	template names .....	207
	template type equivalence .....	207
	template_arg .....	205
	template_name .....	206
C++	templates .....	205
casts implemented as	templates .....	247
class	templates .....	205
function	templates .....	208
multiple declarations using function	templates .....	210
	templates and namespaces .....	224
customizing the behavior of function	terminate() .....	232
function	terminate() .....	232, 235
	ternary operator ?: .....	78
	ternary operators .....	26
hidden argument	this ..... 13, 134, 135, 136, 141, 147, 165	
	this pointer .....	141
	this->class_member .....	135
precedence of the	throw expression .....	226
using the	throw statement without an expression .....	230
	throw statements as conditional operands .....	236
the	throw, try, and catch statements .....	225, 226
	throwing an exception on error .....	225
predefined macro	__TIME__ .....	42
	toggle Long_enums - map type enum variables to an int-sized	
	area of memory .....	275
	token pasting .....	276
	token substitution .....	42
	token_string .....	36, 41
	tokens .....	15
	topmost assignment operator .....	188
	topmost constructor .....	172
	transitive using directives .....	223
iterating through the nodes of a binary	tree .....	283
	tree data structures .....	284
recursive	tree walk .....	283, 284
	trigraphs in C++ .....	32
jumps with	try blocks and exception handlers .....	236
	try statement syntax .....	226
the throw,	try, and catch statements .....	225, 226
allocating an object of array	type .....	91
class	type .....	128
distance-attributed	type .....	270



enumeration	type	275
function return data	type	112
member	type	128
pointer-to-member of a function	type	149
sizeof a full-function	type	266
using keyword typeid to find the complete	type	240
va_list	type	120
	type and assignment compatibility	270
run-time checks for validity of	type casting	249, 250
	type casting	96, 239, 247
syntactic ambiguity of declarations versus	type casts	243
data	type char	56
data	type char []	31
improved	type checking at the iterator invocation	287
parameter	type compatibility and conversion	117
automatic	type conversions	93
dangerous	type conversions	248
implicit	type conversions	93, 97
raw	type conversions	250
data	type double	56
	type enum	58
toggle Long_enums - map	type enum variables to an int-sized area of memory	275
template	type equivalence	207
data	type float	56
void return	type for constructors	256
run-time	type information	239
data	type int	56
data	type long double	56
data	type long int	56
High C/C++ extensions: data	type long long int	56, 57
argument	type matching	194
making	type names visible with typedefs	130
copies of the	type object	241
using the	type object	242
	type object of a polymorphic class	241
functions on	type objects	244
	type objects	240
complete	type of a subobject of a polymorphic class	239, 240
static	type of a subobject of a polymorphic class	240
	type of an assignment expression	80
High C/C++ extensions:	type qualifier _Alias	19, 69
High C/C++ extensions:	type qualifier _Far	19, 269
High C/C++ extensions:	type qualifier _Near	19, 269
High C/C++ extensions:	type qualifier _Noalias	19, 69
	type qualifier const	67
	type qualifier volatile	49, 68

	type qualifiers .....	67
data	type short int .....	56
data	type signed char .....	56
default data	type signed int .....	56
	type specifiers .....	54
conversion from a more precise	type to a less precise type .....	93
data	type unsigned char .....	56
data	type void .....	56
data	type wchar_t .....	30
data	type wchar_t [] .....	31
	type_name .....	91, 96
	type_qualifier .....	67
	type_specifier .....	54
defining types with	typedef .....	69
	typedef members .....	131
	typedef names within structures .....	257
making type names visible with	typedefs .....	130
polymorphic	typeid .....	241
checking for	typeid equality .....	242
using keyword	typeid to find the complete type .....	240
header file	typeinfo.h .....	244, 245
arithmetic data	types .....	56
attributed return	types .....	270
casts of pointer	types .....	271
comparing operands of different	types .....	83
converting to known and unknown	types .....	181
data	types .....	52
declarations of enum	types .....	275
derived	types .....	59
dynamic_cast and reference	types .....	246
floating data	types .....	56
fundamental	types .....	56
High C/C++ extensions: long and short		
enumeration	types .....	59, 275
integral data	types .....	56
modifying	types .....	67
scalar	types .....	59
signed and unsigned data	types .....	56
size of data	types .....	57
using dynamic_cast with pointer	types .....	246
yield	types .....	288
	types a type object can represent .....	245
order and data	types of function parameters .....	117
defining	types with typedef .....	69
	types yielded by an iterator .....	289

## U

	unary minus operator - .....	23, 88
prefix	unary operators .....	201
	unary operators .....	20, 23, 87
postfix	unary operators ++ and -- .....	201
	unary plus operator + .....	23, 87
	unary preprocessor operator defined .....	40
preprocessor directives #assert and	#unassert .....	276
preprocessor directives #define and	#undef .....	36
High C/C++ extensions:	underscores in numbers .....	27, 88, 259
customizing the behavior of function	unexpected() .....	235
class compared with struct and	union .....	62, 123, 126, 128, 138
public access implied by struct or	union .....	153
storage for simple members of a	union .....	129
anonymous	unions .....	65, 188
sizeof value of	unions .....	129
structures and	unions .....	62
	unions .....	64
	uniqueness of names .....	50
	uniqueness of template names .....	207
internal linkage within an	unnamed namespace .....	223
	unnamed namespaces .....	222
High C/C++ extensions: keywords _Packed and	_Unpacked .....	19, 63, 268
High C/C++ extensions: packed and	unpacked structures .....	268
data type	unsigned char .....	56
signed and	unsigned data types .....	56
additional clean-up during	unwinding .....	229
exception handling during	unwinding .....	232
	unwinding .....	228
	up-level addressing .....	263
	up-level referencing .....	263
	user-defined conversions .....	179
ambiguities between	user-defined operators and pre-defined C++ operators ....	182
overloading functions with the	using declaration .....	218
the	using declaration .....	218
	using declarations inside namespaces .....	219
explicit qualification and names introduced	using directive .....	222
by the	using directive .....	220
the	using directives .....	223
transitive	using N::* expression .....	220
the	using-directive syntax .....	220

## V

	va_arg macros .....	289
macros va_start(),	va_arg(), and va_end() .....	120
macros va_start(), va_arg(), and	va_end() .....	120
	va_list type .....	120
macros	va_start(), va_arg(), and va_end() .....	120
function address versus full-function	value .....	264
passing parameters by	value and by reference .....	117
sizeof	value of unions .....	129
! following parameter list for full-function	values .....	264
or nested-function	values .....	79
case	values for an integer .....	26
maximum and minimum	values for enumerators .....	58, 79
initial	variable number of arguments .....	289
	variable-length parameter lists .....	119
for	variables .....	279, 288, 289, 291, 292
High C/C++ extensions: nested functions and	variables .....	263
full-function	variables .....	254
initialization of static	variables .....	267
register	variables .....	47
accessing	variables by indirect means .....	69
static members and	variables in a template .....	212
toggle Long_enums - map type enum	variables to an int-sized area of memory .....	275
reference	variables used in expressions .....	60
multiple access through	virtual base classes .....	158
	virtual base classes .....	146
	virtual base classes and constructors .....	172
	virtual base pointers .....	147
	virtual bases .....	138
	virtual call mechanism .....	145
overriding a	virtual function .....	141
pure declaration of a	virtual function .....	146
semantics of a	virtual function .....	141
constructors and	virtual functions .....	167
pure	virtual functions .....	14, 128
	virtual functions .....	13, 141, 144
	virtual functions versus virtual bases .....	146
sticky	virtual specifier .....	145
	virtual-function tables .....	145, 169, 178
	virtual-function-table pointers .....	145
datatype	void .....	56
pointer to	void .....	95, 96
	void return type for constructors .....	256

casting away	volatile .....	248
functions qualified with	volatile .....	136
keywords const and	volatile .....	113, 271
type qualifier	volatile .....	49, 68
declaring a const or	volatile function .....	255

## W

recursive tree	walk .....	283, 284
array of	wchar_t .....	31
data type	wchar_t .....	30
data type	wchar_t [] .....	31
	while statement .....	106
	whitespace .....	16
	wide characters for international character sets .....	30
	widened object .....	94
	Windows applications .....	273

## X

	X3J11/90-013 .....	1
ANSI	X3J16 standard .....	1, 209

## Y

	yield parameter of an iterator .....	291
	yield types .....	288
calls to function	yield() .....	285, 290
parameters to function	yield() .....	289
predefined function	yield() .....	279, 280, 281, 288
function	yield() formal-parameter list .....	289
types	yielded by an iterator .....	289
results	yielded by the iterator .....	280, 287

